Large-Scale Clone Detection and Benchmarking

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
in Partial Fulfillment of the Requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Jeffrey Svajlenko

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

Code clones are pairs of code fragments that are similar. They are created when developers re-use code by copy and paste, although clones are known to occur for a variety of reasons. Clones have a negative impact on software quality and development by leading to the needless duplication of software maintenance and evolution efforts, causing the duplication and propagation of existing bugs throughout a software system, and even leading to new bugs when duplicate code is not evolved in parallel. It is important that developers detect their clones so that they can be managed and their harm mitigated. This need has been recognized by the many clone detectors available in the literature. Additionally, clone detection in large-scale inter-project repositories has been shown to have many potential applications such as mining for new APIs, license violation detection, similar application detection, code completion, API recommendation and usage support, and so on.

Despite this great interest in clone detection, there has been very little evaluation of the performance of the clone detection tools, including the creation of clone benchmarks. As well, very few clone detectors have been proposed for the large-scale inter-project use cases. In particular, the existing large-scale clone detectors require extraordinary hardware, long execution times, lack support for common clone types, and are not adaptable to target and explore the emerging large-scale inter-project use-cases. As well, none of the existing benchmarks could evaluate clone detection for these scenarios.

We address these problems in this thesis by introducing new clone benchmarks using both synthetic and real clone data, including a benchmark for evaluating the large-scale inter-project use-case. We use these benchmarks to conduct comprehensive tool evaluation and comparison studies considering the state of the art tools. We introduce a new clone detector for fast, scalable and user-guided detection in large inter-project datasets, which we extensively evaluate using our benchmarks and compare against the state of the art.

In the first part of this thesis, we introduce a synthetic clone benchmark we call the Mutation and Injection Framework which measures the recall of clone detection tools at a very fine granularity using artificial clones in a mutation-analysis procedure. We use the Mutation Framework to evaluate the state of the art clone detectors, and compare its results against the previous clone benchmarks. We demonstrate that the Mutation Framework enables accurate, precise and bias-free clone benchmarking experiments, and show that the previous benchmarks are outdated and inappropriate for evaluating modern clone detection tools. We also show that the Mutation Framework can be adapted with custom mutation operators to evaluate tools for any kind of clone.

In the second part of this thesis, we introduce BigCloneBench, a large benchmark of 8 million real clones in a large inter-project source datasets (IJaDataset: 25K projects, 250MLOC). We built this benchmark by mining IJaDataset for functions implementing commonly needed functionalities. This benchmark can evaluate clone detection tools for all types of clones, for intra-project vs inter-project clones, for semantic clones, and for clones across the entire spectrum of syntactical similarity. It is also the only benchmark

capable of evaluating clone detectors for the emerging large-scale inter-project clone detection use-case. We use this benchmark to thoroughly evaluate the state of the art tools, and demonstrate why both synthetic (Mutation Framework) and real-world (BigCloneBench) benchmarks are needed.

In the third part of this thesis, we explore the scaling of clone detection to large inter-project source datasets. In our first study we introduce the Shuffling Framework, a strategy for scaling the existing natively non-scalable clone detection tools to large-scale inter-project datasets, but at the cost of a reduction in recall performance and requiring a small compute cluster. The Shuffling Framework exploits non-deterministic input partitioning, partition shuffling, inverted clone indexing and coarse-grained similarity metrics to achieve scalability. In our second study, we introduce our premier large-scale clone detection tool, CloneWorks, which enables fast, scalable and user-guided clone detection in large-scale inter-project datasets. CloneWorks achieves fast and scalable clone detection on an average personal workstation using the Jaccard similarity coefficient, the sub-block filtering heuristic, an inverted clone index, and index-based input partitioning heuristic. CloneWorks is one of the only tools to scale to an inter-project dataset of 250MLOC on an average workstation, and has the fastest detection time at just 2-10 hours, while also achieving the best recall and precision performances as per our clone benchmarks. CloneWorks uses a user-guided approach, which gives the user full control over the transformations applied to their source-code before clone detection in order to target any type or kind of clones. CloneWorks includes transformations such as tunable pretty-printing, adaptable identifier renaming, syntax abstraction and filtering, and can be extended by a plug-in architecture. Through scenarios and case studies we evaluate this user-guided aspect, and find it is adaptable has high precision.

Acknowledgements

I would like to express my heartfelt appreciation to my supervisor, Dr. Chancel Roy, for his constant and continued support, guidance, encouragement and patience during my graduate studies.

I would like to thank Dr. Michael Horsch, Dr. Nadeem Jamali, and Dr. Aryan Saadat Mehr for their participation on my supervising committee, including their advisement and evaluation of my work and thesis. I would like to thank my external examiner Dr. Nikolaos Tsantalis for his evaluation of my thesis.

I would like to thank the co-authors of the papers I have published during my graduate studies, including Chanchal K. Roy, Iman Keivanloo, Hitesh Sajnani, Judith F. Islam, Mohammad Mamun Mia, James Cordy, Vaibhav Saini and Cristina V. Lopes.

I would like to thank all of present and past members of the Software Research Lab, who have given me their support, advise and friendship on many occasions. As well for the many hours of clone validation efforts they have volunteered for my research studies.

I would like to thank the faculty and staff of the Department of Computer Science and College of Graduate Studies for their support over the years. In particular, I would like to thank Gwen Lancaster.

I am grateful for the financial support, including scholarships and awards, provided by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Department of Computer Science of the University of Saskatchewan and the College of Graduate Studies of the University of Saskatchewan, which allowed me to focus on my studies and this thesis work.

I give my heartfelt thanks to my mother Donna Svajlenko and my father Shane Svajlenko for their unconditional support and encouragement throughout my graduate studies. They celebrated with me through the highs, and motivated and pushed me through the lows, and I really owe the completion of this thesis to them.

For my parents, Donna and Shane Svajlenko.

Contents

List of Tables

List of Figures

List of Abbreviations

UPI	unique percentage of items
LOC	lines of code
KLOC	thousand lines of code
MLOC	million lines of code
API	application program interface
T1	Type-1
T2	Type-2
T3	Type-3
T4	Type-4
VST3	Very Strongly Type-3
ST3	Strongly Type-3
MT3	Moderately Type-3
WT3/T4	Weakly Type-3 or Type-4

Chapter 1

Introduction

Code clones, or software clones, are pairs of code fragments within a software system that are similar. Code reuse by copy and paste, with or without modifications, is one of the common sources of code clones, although they are known to arise for a variety of reasons [2, 8, 10, 55, 58, 72, 88, 104, 108, 109]. Software developers frequently employ copy and paste code re-use as it is faster and easier than developing new code or abstracting existing code [11,68,82]. Unfortunately, cloning also has a negative impact on software quality and software development costs. Cloning leads to an unnecessary increase in the size of the code base, which needlessly increases the cost of software maintenance and evolution tasks that scale with the size of the code base [88, 92]. Cloning harms software quality by increasing the number of bugs in the system [50, 82]. When an existing code fragment contains a bug, cloning it duplicates and propagates the bug across the system. If a code fragment is evolved or otherwise modified, and those modifications are not appropriately and correctly propagated to its duplicates, a bug is created as a result of cloning [82, 83]. As cloning is not usually documented, and the developer making the change to a code fragment may not be the one who previously cloned the code fragment, inconsistent changes to duplicated code is not atypical and therefore there is a real risk of cloning related bugs [103]. While cloning can also have benefits, such as accelerated software development and increased decoupling [26, 59, 109], it is important that developers keep track of their clones in order to manage and reduce their negative effects [78].

Clone detection tools have been developed to locate clones within or between software systems [104, 108]. These tools provide awareness of the clones within a software system, which allows developers to mitigate their harm on development costs and software quality. Developers can mitigate the harmfulness of a clone by removing it using software refactoring and re-engineering (e.g., extracting the shared code into an abstraction mechanism) [89]. In cases where refactoring is too expensive or difficult (e.g., social factors such as code ownership), the awareness of these clones provided by the clone detectors can be used to monitor and prevent the potential harm [95, 106]. When a code fragment is modified to fix a bug, clones of that code fragment should be inspected for the same bug. Similarly, clones of an evolved code fragment can be inspected to see if the evolution needs to be propagated to the duplicates. There are four primary clone types, including syntactically identical clones (Type-1), structurally identical clones (Type-2), syntactically similar clones (Type-3) and semantically similar but syntactically dissimilar clones (Type-4). Most clone detection tools target the detection of up to type Type-3 clones within a single software system.

While clone detectors were created to aid in the refactoring and management of code clones, a number of other applications have been demonstrated. The detection of similar code has been shown to be useful in plagiarism detection [43, 79, 100], license violation detection [74], origin analysis [42], software evolution analysis, multi-version program analysis [69], bug detection [51, 54, 82], aspect mining [17], program comprehension [56], code compaction [30], malware detection [138], software product line analysis [28], and so on [104, 108, 109].

An emerging topic in clone research is the detection of clones within very-large inter-project source code datasets containing on the order of tens of thousands of software projects or more. This has many research applications, including: studying global open-source developer practices [96], building new application programming interfaces (APIs) [48], license violation detection [73], similar mobile application detection [22], large-scale clone and code search [61, 81], code completion [49], API recommendation and usage support [66], and so on. These applications require clone detectors that scale to hundreds of millions of lines of code or larger. Large-scale clone detection is needed for migrating software variants towards a software product line [45], to detect clones in large software ecosystems (e.g., Debian), and to study cloning in the global open-source community (e.g., GitHub). The detection of code smells such as clones is very important in critical systems, such as modern automobile systems, which can contain up to 100 million lines of code [3], while large software companies have software portfolios reaching a billion lines of code [91].

Clone detection tools are evaluated using information retrieval metrics including recall and precision [13]. Recall is the ratio of the clones within a software system that a tool is able to detect, while precision is the ratio of the clones reported by a clone detector that are actually clones and not false positives. Precision can be measured by manually validating a random sample of the clones a tool detects across a variety of subject systems. Measuring precision is simple but very time consuming. Recall is very challenging to measure as it requires independent knowledge of the clones that exist in a subject software system(s). Building high quality clone benchmarks of known reference clones to measure recall has been very difficult for the clone research community.

1.1 Research Problem

A 2009 survey of clone detection tools and techniques by Roy et al. [108] found the existence of at least 39 clone detection tools. A 2013 survey by Rattan et al. [104] found at least 70 tools, a 75% increase in only four years. Our literature review in 2017 found at least 198 tools, a 182% increase in the four years following Rattan's study. This number reflects the importance the community has placed on clone detection for use in software development and research studies. However, I note two significant deficits in the state of clone detection research.

First, despite the emerging applications of clone detection in large inter-project source-code datasets, there is a lack of tools that can scale to such large inputs, in particular for Type-3 clones. Additionally,

there is no user-guided tools in this domain, which can be configured to detect any type or kind of clone, or be easily extended to detect novel kinds of clones. This lack makes pursuing the emerging applications in large-scale and inter-project clone detection difficult. Therefore, researchers must develop domain-specific clone detection tools for their studies, which takes significant effort. These tools have significant limitations beyond the specific study, so are not suitable for re-use. They are not user-guided in detecting different types and kinds of clones, and require significant computing resources. The community would benefit from a fast, scalable and user-guided clone detection tool that can be re-used across various studies, freeing the researcher from developing a whole new tool for their task, and allowing them to focus on the goals of their study.

Second, despite the large number of published clone detection tools and techniques, there is a lack of clone detection benchmarks and standard tool evaluation procedures. As a consequence, there has also been a lack of tool evaluation and comparison studies. Clone detection tools are evaluated for their recall, precision, execution time and scalability. While precision, execution time and scalability can be measured by the tool authors without external support, measuring recall requires a clone benchmark of known reference clones. Therefore, tool authors rarely measure recall, and often they either do not measure the other metrics or do a poor job in measuring them. This has led to the situation where so many clone detection tools are available in the literature, but there is very little empirical evidence of their performance. Researchers do not know which tools to use, and instead create new tools for their research. There is also a strong preference for people to use familiar tools, such as those produced by their own lab or which have been historically popular in clone studies, instead of the best tool for their use-case. Therefore, authors rarely even release research prototypes of their tools, as tool adoption is difficult. Needed are high quality and easy to use clone benchmarks for measuring recall, tool comparisons comparing the state of the art and popular tools, and tool evaluation studies that demonstrate standard methods for measuring the four clone detection performance metrics. Additionally, to deliver a flexible clone detection tool for large-scale clone detection, I need benchmarks designed to evaluate large-scale clone detection, including for the inter-project use-cases.

These research problems are strongly interconnected. In order to introduce a new large-scale clone detection tool, I need clone benchmarks to demonstrate its performance and justify its value. Clone benchmarks are needed by the community to drive real improvements in the clone detection tools backed by empirical evidence. As well, in order to demonstrate new benchmarks targeting emerging clone detection applications, such as large-scale inter-project clone detection, I need accessible large-scale clone detection tools designed for this domain.

In summary, there are a number of significant deficits in the clone detection tool literature:

1. A lack of fast, scalable and user-guided clone detection tools for clone detection in large inter-project source datasets, in particular for Type-3 clones and emerging clone types.

2. A lack of modern and high-quality clone benchmarks for measuring clone detection recall, and in particular for the emerging large-scale inter-project clone detection domain.

3. A lack of empirical studies comparing the performance of state of the art clone detection tools, in particular their recall, precision, execution time and scalability performances.

1.2 Addressing the Research Problems

To address these problems, I perform research to contribute towards the advancement of large-scale inter-project clone detection and clone detection tool benchmarking and comparison.

To address the problems in clone detection tool evaluation I introduce two modern clone detection benchmarks: the Mutation and Injection Framework and BigCloneBench. The Mutation and Injection Framework is a synthetic benchmark which measures recall at a very fine granularity in a mutation-analysis procedure. The Mutation Framework automates controlled and biased-free recall measurement experiments. BigCloneBench is a big benchmark of over 8 million real clones in a large (25K projects, 250MLOC) inter-project source-code dataset. It can measure recall from a variety of perspectives, including: per clone type, for inter-project vs intra-project clones, for semantic and syntactic clones, and across the entire spectrum of syntactical similarity. In particular, BigCloneBench targets the evaluation of clone detectors for large-scale inter-project clone detection. With these benchmarks, I deliver a complete clone detection recall measurement procedure which combines the best of synthetic and real-world clone benchmarking techniques, which I find to be essential to accurately measure recall.

I use our benchmarks in a number of tool evaluation and comparison studies, which measure the performance of the state of the art tools. I use our Mutation Framework to evaluate the recall and capabilities of the tools at a fine granularity and for particular kinds of clones. I use our BigCloneBench to measure the recall of the tools for real clones, including inter-project vs intra-project clones, and across the entire spectrum of clone types and syntactical similarity. I compare the Mutation Framework and BigCloneBench results, and justify the need for both synthetic and real-world benchmarking. Using our benchmarks, I evaluate state of the art tools, measuring their recall, precision, execution time, and scalability up to and including a large inter-project source-code datasets (250MLOC). I provide the most comprehensive analysis of modern clone detection tool performance, and demonstrate a complete tool evaluation procedure that can be reused by the community as a standard methodology. Before our work, the most widely accepted clone benchmark was Bellon's Benchmark. In a quantitative study, I demonstrate that it may not appropriate for evaluating the modern clone detection tools, demonstrating the need for our new and modern clone benchmarks. This finding is supported also by the related work studying Bellon's Benchmark [9, 19, 21].

To address the lack of clone detection tools targeting large-scale inter-project clone detection, I introduce a new clone detection tool: CloneWorks, and evaluate it using our clone benchmarks. I begin with an exploratory study evaluating heuristics for scaling existing clone detection tools to large-scale. I find that input partitioning and clone indexes are a promising method of scaling clone detection to large scale on standard hardware. I combine input partitioning and clone indexing with efficient clone similarity metrics

and filtering heuristics from the literature to build a clone detector with the best execution time and scalability on commodity hardware. I design a user-guided parsing and source transformation pipeline, with plug-in support, that enables users to target any clone type, as well as explore new kinds of clones using custom source normalization and processing. I demonstrate CloneWorks's user-guided approach in a number of case studies, and then evaluate it against the state of the art using our clone benchmarks. BigCloneBench shows that CloneWorks is the best tool for large-scale clone detection experiments.

1.3 Decomposing our Research behind Addressing the Problem

I decompose our research into the following three parts:

- Part 1 - Synthetic Clone Benchmarking with Mutation Analysis.
- Part 2 - Real-World Large-Scale Clone Benchmarking.
- Part 3 - Large-Scale Clone Detection.

1.3.1 Part 1 - Synthetic Clone Benchmarking with Mutation Analysis

In this part of the thesis, I present our work using mutation analysis and source-code injection to build synthetic clone benchmarks and to evaluation the recall of clone detection tools. I begin by introducing our Mutation and Injection Framework (Chapter 3), which evaluates clone detectors in a mutation-analysis procedure. It synthesizes reference clones using fifteen mutation operators based on a taxonomy of the types of edits developers make on copy and pasted code, and automates the execution of the subject clone detection tools for the benchmark and measures their recall at a fine granularity. I use this framework to generate large Java and C clone benchmarks and use them to evaluate the state of the art tools (Chapter 4). I compare these results against the previous leading clone benchmark, Bellon's Benchmark [13,14], to show that it is not appropriate for evaluating the modern clone detection tools. In this study I demonstrate the accuracy of the Mutation Framework and motivate the need for a new real-world benchmark to replace Bellon's Benchmark (which I address in Part II). I then use the benchmark to extensively evaluate the modern clone detection tools at a very fine granularity (per type of clone edit from the taxonomy) for Java, C and C# clones, and use the results to pin-point their strengths and weaknesses (Chapter 5). The Mutation Framework is designed to be extensible, so that users can produce clone benchmarks for evaluating recall for any type or kind of clone. I demonstrate this by designing mutation operators for creating gapped clones (identical code fragments, except for the insertion of a sequence of dissimilar source lines into one copy) for various gap lengths. I generated a corpora of gapped clones, and evaluate the robustness of the clone detection tools against clones with gaps of different lengths (Chapter 6). I also show how our mutation and injection technology can be adapted to benchmark other kinds of software analysis tools. I adapt our technology to build synthetic

datasets of software variants (i.e., forks) with known similarities and differences to evaluate software variant analysis tools (e.g., clone detectors) that support migration towards software product lines (Chapter 7).

1.3.2 Part 2 - Real-World Large-Scale Clone Benchmarking

In this part of the thesis, I present our real-world large-scale clone benchmark, BigCloneBench. I begin by describing the creation of BigCloneBench (Chapter 8), where I used a novel and scalable clone mining and validation procedure to discover, without the use of clone detectors, function clones similar by their implementation of specific functionalities. I built a benchmark of over 8 million reference clones implementing 48 distinct functionalities that span the entire range of syntactical similarity including both intra-project and inter-project clones. I use this benchmark to perform a thorough evaluation of the state of the art clone detection tools (Chapter 10), and measure their recall for all four of the primary clones, for intra-project vs inter-project clones, and across the entire spectrum of syntactical similarity. I compare these results against those from our Mutation and Injection Framework to justify the need for both synthetic and real-world benchmarks. I distilled our tool evaluation procedure into an evaluation framework called BigCloneEval (Chapter 10), which makes it easy for the the community to reproduce, extend and customize our tool comparison experiment with BigCloneBench. In particular, BigCloneEval handles the complexity of working with such a large clone benchmark and dataset, and automates recall evaluation experiments.

1.3.3 Part 3 - Large-Scale Clone Detection

In this part of the thesis, I present our work on large-scale clone detection. I begin with an exploratory study using our Shuffling Framework (Chapter 11) where I investigate the use of non-deterministic input-partitioning, clone indexing, and coarse-grained code similarity metrics to scale existing non-scalable clone detection tools. I successfully scale these tools to a large inter-project dataset (25K projects, 250MLOC) with an acceptable loss of recall and by distribution over a small cluster. I use our experience with the Shuffling Framework to create a new fast, scalable and user-guided clone detector called CloneWorks (Chapter 12). CloneWorks detects clones using the efficient Jaccard similarity coefficient. Fast execution time is achieved by a fully in-memory clone indexing approach with the sub-block filtering heuristic introduced in our published work [116]. Scalability within available memory is achieved using an index-based input partitioning scheme inspired by our Shuffling Framework. A user-guided approach is achieved by a customizable and pluggable source-code parsing and transformation pipeline, which enables the user to target any type or kind of clones as needed for their scenario or user-case. I evaluate CloneWork's detection performance and compare against the state of the art tools using our Mutation Framework (Part I) and BigCloneBench (Part II), and find it can achieve best-in-class recall and precision performance. CloneWorks is the fastest tool to scale to a large inter-project dataset (25K projects, 250MLOC) on a single personal computer, just two to ten hours of execution time depending on the configuration. I evaluate the user-guided approach by detecting clones with various source transformations targeting different scenarios and use-cases, including a novel API usage

clone detection. As part of this evaluation I manually validate over 15K clone pairs to measure the precision of the user-guided approach, which is the most extensive precision evaluation of any clone detector.

1.4 Outline of the Thesis

- **Chapter 1** introduces the research problem and how I addressed it.
- **Chapter 2** provides essential background knowledge.
- **Part I** describes our research developing, evaluating and using our synthetic clone benchmark: The Mutation and Injection Framework.
  - **Chapter 3** describes the Mutation and Injection Framework methodology and procedure.
  - **Chapter 4** contains our study using the Mutation Framework to evaluate state of the art tools, and to evaluate the state of evaluation of modern clone detection tools, including the evaluation of a previous benchmark: Bellon's Benchmark. In this study I justify the need for our synthetic benchmark, and the need for a new real-world clone benchmark (addressed by our BigCloneBench).
  - **Chapter 5** contains our study using the Mutation Framework to evaluate the state of the art tools at a fine granularity.
  - **Chapter 6** contains our study using the Mutation Framework to evaluate state of the art tools for gapped clones with gaps of differing sizes. In this way, I demonstrate the extensibility of the Mutation Framework with custom clone-producing mutation operators.
  - **Chapter 7** describes ForkSim, our framework for developing synthetic forks for benchmarking software variant analysis tools (including clone detectors). In this way, I demonstrate how our source-code mutation and injection technology can be used to build benchmarks for related software engineering fields.
- **Part II** describes our research developing and using our real-world and large-scale clone benchmark: BigCloneBench.
  - **Chapter 8** describes the creation of BigCloneBench, and how it can be used to evaluate clone detection tools.
  - **Chapter 9** contains our study using BigCloneBench to evaluate state of the art tools, for intra-project vs inter-project clones and across the spectrum of syntactical similarity, including demonstrating the need for both real-world and synthetic clone benchmarks.
  - **Chapter 10** describes our recall evaluation framework, BigCloneEval, built on top of BigCloneBench and implementing a customizable recall evaluation procedure.

- **Part III** describes our research exploring and developing clone detection tools for large-scale clone detection experiments.
  - **Chapter 11** contains our study exploring procedures and heuristics for scaling the classical non-scalable clone detection tools to large inter-project datasets.
  - **Chapter 12** describes CloneWorks, our fast and flexible clone detection tool for large-scale clone detection experiments.
- **Chapter 13** concludes our work, including a summary of our publications, and directions for future research.

1.5 Manuscript-Style Thesis

The remainder of this thesis is written in the manuscript style. The chapters have been created using published and unpublished (in-submission or to be submitted to an academic conference or journal) with some reformatting and editing to fit this thesis. In the introduction to each chapter, I indicate the source of the manuscript, if and where it has been published, and acknowledge the contributions of any co-authors. For all of the included works I was the lead researcher and the manuscripts are written by myself as the lead author. In the included manuscripts, co-authors have either taken a supervisor role for the research, or contributed towards clone validation efforts. With respect to my supervisor, Chanchal Roy, and my co-authors, I used the pronoun "we" for the remainder of this thesis, except when explicitly referring to myself.

The manuscripts forming chapters in this thesis, and their copyright, are as follows:

- J. Svajlenko and C. K. Roy, "Evaluating Modern Clone Detection Tools," 2014 IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, 2014, pp. 321-330. ©2014 IEEE. (Chapter 4).
- J. Svajlenko, C. K. Roy and S. Duszynski, "ForkSim: Generating software forks for evaluating cross-project similarity analysis tools," 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM), Eindhoven, 2013, pp. 37-42. ©2013 IEEE. (Chapter 7).
- J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy and M. M. Mia, "Towards a Big Data Curated Benchmark of Inter-project Code Clones," 2014 IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, 2014, pp. 476-480. ©2014 IEEE. (Chapter 8).
- J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with BigCloneBench," 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), Bremen, 2015, pp. 131-140. ©2015 IEEE. (Chapter 9).

- J. Svajlenko and C. K. Roy, "BigCloneEval: A Clone Detection Tool Evaluation Framework with BigCloneBench," 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), Raleigh, NC, 2016, pp. 596-600. (Chapter 10).

- J. Svajlenko, I. Keivanloo, and C. K. Roy, "Big data clone detection using classical detectors: an exploratory study," June 2015, Journal of Software: Evolution and Process, Volume 27, Issue 6, 430464. (Chapter 11).

Chapter 2

Background

2.1 Cloning Theory

Code clones are instances of similar code fragments, for some definition of similarity. Generally, we are interested in code fragments that are textually, syntactically or functionally similar, although there is no restriction on the kind of similarity in the clone definition. Code clones are typically documented as clone pairs or clone classes. The following are standard definitions for these terms [109]:

Code Fragment (or Code Snippet) A contiguous region of source code within a source file. Specified by the triple (f, s, e), including the source file f, the line the code fragment starts on s, and the line it ends on e.

Clone Pair A pair of code fragments that are similar, for some definition of similarity. Specified by the tuple (f_1, f_2, τ), including the similar code fragments f_1 and f_2 and their type of similarity τ.

Clone Class A set of code fragments that are similar. Specified by the tuple $(f_1, f_2, ..., f_n, \tau)$, including the n similar code fragments and their type of similarity.

Clone pairs may be reported with or without the type of similarity, τ, explicitly indicated. Additionally, some clone detectors mix clones of various types within the same clone class. The base requirement of a clone class is that every pair of code fragments within the class form a valid clone pair.

2.1.1 Clone Types

Researchers agree upon four primary clone types that are mutually exclusive and defined with respect to the detection capabilities needed to detect them [109]:

Type-1 Identical code fragments, ignoring differences in white-space, code formatting/style and comments.

Type-2 Structurally/syntactically identical code fragments, ignoring differences in identifier names and literal values, as well as differences in white-space, code formatting/style and comments.

Type-3 Syntactically similar code fragments, with differences at the statement level. The code fragments have statements added, removed or modifiers with respect to each other.

Type-4 Syntactically dissimilar code fragments that implement the same or similar functionality.

The Type-1 and Type-2 clone types are well defined, while the Type-3 and Type-4 clone types are more open to individual interpretation. While researchers agree upon these definitions, they may disagree upon what is the minimum syntactical similarity of a Type-3 clone, or the minimum functional similarity of a Type-4 clone.

2.1.2 Type-1 Clones

Type-1 clones are pairs of identical code fragments, when we ignore trivial differences in extraneous white-space, code formatting/style and commenting. Type-1 clones can occur when a code fragment is copy and pasted, with only trivial modifications to match the destination's formatting and code style, perhaps with the addition, removal or modification of comments. Clone detectors detect Type-1 clones by parsing the source-code in a way that removes or normalizes the allowed differences, and detects those code fragments that become textually or syntactically identical. For example, a clone detector can tokenize the source code, and search for the identical token sequences to detect Type-1 clones. Tokenization retains only the language tokens, and drops the commenting, white-space and formatting. In some languages, such as Python, some white-space are tokens as they have syntactical meaning beyond token separation.

2.1.3 Type-2 Clones

Type-2 clones are pairs of code fragments that are syntactically identical when we ignore differences in identifier names and literal values, in addition to white-space, layout, style and comments. Type-2 clones can occur when a developer has reused a code fragment by copy and paste, and has renamed one or more variables to better match the destination. In addition to renamed variables, Type-2 clones can have renamed/changed constants, classes, method names, and so on; as well as changes in literal values and types. Type-2 clones can be detected by normalizing identifier names and literal values, in addition to the Type-1 normalizations, and detecting syntactically or textually identical normalized code fragments. For example, a clone detector could tokenize the code, and replace each identifier token with a common token (e.g., 'identifier') and the same for each literal token (e.g., 'literal'), and detect the token sequences that are identical as Type-2 clones.

2.1.4 Type-3 Clones

Type-3 clones are those that contain line-level (statement) differences, with the code fragments containing lines added/removed or modified with respect to each other. Type-3 clones can occur when a code fragment has been duplicated and then modified at the line-level to satisfy new requirements. The duplicated code fragment could be extended with new statements to add a new feature, statements could be removed to

remove an unneeded feature, or statements could be modified to adjust an existing feature for the new use-case. The Type-3 clone definition also allows for Type-1 and Type-2 differences to occur between the code fragments. Clone detectors can detect Type-3 clones using clone similarity metrics that measure the syntactical similarity of two code fragments, perhaps after Type-1 and Type-2 normalizations, then reporting those that satisfy a given minimum similarity threshold. Another method is to detect nearby Type-1/Type-2 clones that are separated by a dissimilar gap, and merge these to form Type-3 clones. Researchers do not agree on how much modification can be performed on the copied code fragment before it is no longer a clone of its original. Since clones can arise for reasons other than copy and paste, for example programming language limitations and repeated coding styles [109], researchers are concerned with what is the minimum syntactical similarity of a valid Type-3 clone.

2.1.5 Type-4 Clones

Type-4 clones can occur when the same functionality has been implemented multiple times using different syntactic variants. Most programming languages allow the same functionality to be specified using different syntax. For example, a switch statement could be replaced with an if-else chain, or a for-loop could be replaced by a while loop. Often the statements of a code fragment can be re-ordered without changing functionality but significantly varying the code fragment's syntax and structure. As a more extreme example, two implementations of merge-sort, one recursive and one iterative, could be considered as a Type-4 clone. Type-4 clones are a relatively unexplored clone type, with very few tools targeting their detection. It is also difficult to separate the Type-3 and Type-4 clones as many Type-3 clones have the same or share functionality, while Type-4 clones will often share some degree of syntactical similarity. There is also the question of how similar must the functionality of two code fragments be for them to be a Type-4 clone, or if implementations of the same functionality (e.g., stable sort) using different algorithms (e.g., bubble sort and merge sort) is a Type-4 clone.

2.1.6 Clone Granularity and Boundaries

The general clone definition does not place any constraints on the boundaries of a code fragment except that it is a contiguous sequence of lines. Therefore, clones can exist at various granularities in source code [109,143]:

File Clone A pair of similar source files.

Class Clone A pair of similar class definitions (in object-oriented code).

Function Clone A pair of similar functions (or methods, constructors, deconstructors, and so on).

Block Clone A pair of similar code blocks (indicated by a matching pair of opening and closing braces, or a sequence of statements at the same indentation, and so on and depending on the programming language syntax style).

Arbitrary Clone A pair of similar code fragments, whose start and end lines do not necessarily align to any well-defined syntax unit.

File, class, function and block clones have precise boundaries. The code fragments start and end on the boundaries of the respective syntax units. For example, a code fragment of a function clone starts on the line the function definition begins on and ends on the line the function definition ends on.

Arbitrary clones have source lines that do not align to any specific source unit. For example, they could be an arbitrary sequence of source lines within a function, or they could start in the preamble of a source file and end in the middle of a function. However, a number of rules for high-quality arbitrary clone reporting are followed by many tools: (1) the code fragments of an arbitrary clone should not overlap, (2) the start and end lines should be within the same scope, (3) the start and end lines should not span multiple functions as function definition order is not relevant in most languages. These are general rules for high-quality arbitrary clone reporting for human inspection and software maintenance tasks. Not all clone detection tools follow these guidelines, often because they can be difficult or computationally expensive to enforce. Additionally, there may be domain-specific applications of clones that violate these rules. For example, clone detection for code compaction using macros.

2.1.7 Clone Size

The clone definitions do not put any constraints on the size of clones. Clone size is typically measured as the maximum (or average) length of its code fragments measured in original source lines, pretty-printed source-clones and/or by token. While there is no minimum size of a clone, very small clones are often spurious. For example, a pair of identical tokens are not typically considered a clone, neither are pair of identical simple statements. Clone detection tools typically require a minimum clone size configuration in order to filter those smaller identical/similar code fragments that are likely to be spurious or uninteresting. Typical minimum clone sizes are 6-15 original source lines [13, 110, 121, 128], and 30-100 tokens [41, 58].

2.1.8 Clone Detection Tools

Clone detection tools are used to detect clones within source code. Given a collection of source code, and a configuration of its algorithms, the clone detector outputs the set of clones it detected in the collection of source code.

The collection of source code can be a single software system, a collection of software systems, or just a loose collection of source files. The tool configuration can include the language(s) of the source files to process, the granularity(ies) to report clones at (e.g., arbitrary, block, function or file), the minimum and maximum sizes of clones to report, the source normalizations to apply during parsing, the similarity threshold or maximum gap size for reporting Type-3 clones, or any other configuration specific to the tool's detection algorithms and implementation. The tool outputs the detected clones as a collection of clone pairs or clone

classes in a clone detection report. There is no universal standard for clone detection report format or style.

All clone detection tools implement the following abstract procedure. The source code files are parsed, and all of the code fragments are identified, subject to a minimum and maximum code fragment size (by line or token), granularity, boundary constraints, file filters, sliding windows, and so on. Let $F = \{f_1, f_2, ..., f_n\}$ be the n code fragments found in the source code, after filtering.

This implies a set of *potential* clone pairs, $F \times F$, that the tool should investigate. The tool may reject some potential clone pairs, such as pairs of the same code fragment, pairs of overlapping code fragments, pairs of code fragments that vary too significantly in size, and so on. The remaining potential clones pairs are investigated by the tool's detection algorithms and similarity metrics to decide if they are a clone or not. The tool outputs its *detected clones* (as clone pairs or summarized in clone classes) – the candidate clone pairs its algorithms have judged to be clones. This is summarized in Eq. 2.1, where judge() either accepts a potential clone pair as a true clone, or rejects it as a false clone, as per the tool's judgment.

$$D = \{(f_i, f_j) \in F \times F \mid i \neq j \wedge judge(f_i, f_j)\} \tag{2.1}$$

The tools generally do not implement this abstract procedure explicitly, but their algorithms efficiently implement this procedure implicitly. Surveys by Roy et al. [108] and Rattan et al. [104] found at least 70 clone detection algorithms and tools in the literature. These have been classified into various categories based on their detection algorithms [108], including: text, token, tree, metric, hash, program dependency graph (PDG), hybrid, and so on.

The large number and variety of detection techniques motivates the need for clone benchmarks. Clone detection tools are not perfect, and their detection reports can contain both true positives and false positives. Additionally, their reporting of a clone's boundaries may not be precise. It may include additional code not part of the clone, or miss some code that is part of the clone.

2.2 Benchmarking Clone Detection

Clone detection performance is measured using information retrieval metrics including recall and precision. Recall is the ratio of the clones within a software system that a clone detector is able to detect, while precision is the ratio of the clones reported by a clone detector that are actually clones and not false positives. Measuring recall requires a clone benchmark, a set of known clones within a subject system(s), which is challenging to build. Precision can be measured without a clone benchmark, but requires extensive clone validation efforts.

2.2.1 Measuring Recall and Precision with an Oracle

Recall and precision can be measured for a specified subject software system as shown in Figure 2.1. Here we have the universe, **U**, of all potential clone pairs (every possible pair of code fragments) within the subject software system. On the left is the set of all true clone pairs, **T**, in the subject software system. This set is

determined by a clone oracle, a hypothetical entity that is perfectly able to judge if a pair of code fragments is actually a clone (in reality, no such entity exists). The remaining pairs of code fragments, $\mathbf{F} = \mathbf{U} - \mathbf{T}$, is the set of false clone pairs – the code fragment pairs the oracle decided are not clones. Then there is the set of detected clone pairs, $\mathbf{D}$, reported by the clone detector when executed for the subject software system. From the perspective of clone detection, this splits the universe into four regions:

True Positives The true clones successfully detected by the subject clone detection tool. (Desirable, improves recall.)

False Positives The false clone pairs incorrectly identified as true clone pairs by the subject clone detection tool. (Undesirable, harms precision.)

True Negatives The false clone pairs that are (correctly) not reported by the subject clone detection tool. (Desirable, improves precision.)

False Negatives The true clone pairs that are not detected (missed) by the subject clone detection tool. (Undesirable, harms recall).

Recall, as shown in Eq. 2.2, is the ratio of the true clone pairs that are detected by the subject clone detection tool, i.e., the ratio of $\mathbf{T}$ that is intersected by $\boldsymbol{D}$. This is also the ratio of the subject tool's true positives to the union of its true positives and false negatives. Therefore, to improve recall, a clone detection tool wants to maximize its true positives and minimize its false negatives.

$$recall = \frac{|\mathbf{D} \cap \mathbf{T}|}{|\mathbf{T}|} = \frac{|true\ positives|}{|true\ positives \cup false\ negatives|} \tag{2.2}$$

Precision, as shown in Eq. 2.3, is the ratio of the detected clone pairs that are true clone pairs, not false clone pairs, i.e., the ratio of $\mathbf{D}$ that is intersected by $\mathbf{T}$. This is also the ratio of the subject tool's true positives to the union of its true positives and false positives. Therefore, to improve precision, a clone detection tool wants to maximize is true positives and minimize its false positives.

$$precision = \frac{|\mathbf{D} \cap \mathbf{T}|}{|\mathbf{D}|} = \frac{|true\ positives|}{|true\ positives \cup false\ positives|} \tag{2.3}$$

2.2.2 Challenges in building an Oracle

The measurement of recall and precision depends on the identification, or "oracling", of all the true clone pairs within a subject system. This process is extremely effort intensive, as it requires the manual examination of every possible pair of code fragments in a subject system. Even a small system such as `cook` (51KLOC, 1244 functions) contains on the order of one million code fragment pairs at the function granularity alone [137]. Not

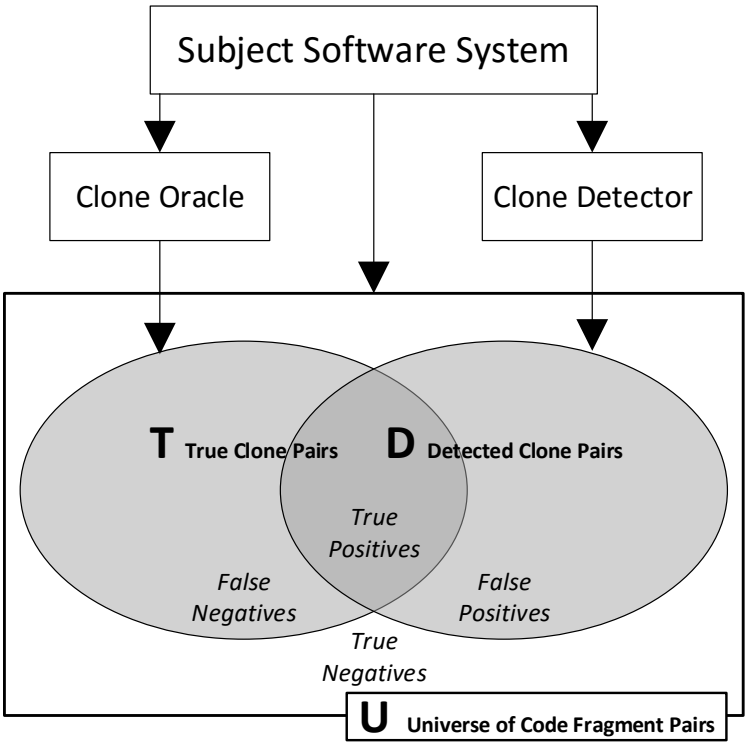


Figure 2.1: Measuring Recall and Precision with an Oracle

only is this too many potential clones to be examined, the `cook` subject system does not contain a sufficient number and variety of true clone pairs on its own to properly evaluate clone detection recall. Additional subject systems are required, which adds to the workload issue.

Additionally, the classification of a potential clone as true or false clone is a subjective process. Previous studies [19] have demonstrated that even among clone experts there is disagreement on what constitutes a true or false clone. A clone expert may even give a different opinion on the same clone when shown it at different times. There is no universal definition of a true clone, and responses might depend on the current task and goals of the clone-detection user [137]. Therefore, it is not only a question of if a potential clone is a true clone, but if it is useful or relevant clone for some clone-related maintenance or development task.

As such, no clone oracle exists, as it is too effort intensive to create. Likely no true oracle can exist, due to the subjectivity in what constitutes a true clone. Instead, clone detection researchers must come up with innovative ways to create corpora of convincingly validated reference clones that can accurately estimate the recall and precision of clone detection tools without the need to fully oracle multiple subject systems. To overcome subjectivity, benchmarks must be created with a well-defined perspective and scope such that the results can be properly interpreted.

2.2.3 Measuring Recall with a Reference Corpus

Recall can be estimated using a *reference corpus*, a set **R** of the known true clone pairs within a subject system, set of subject systems, or some other collection of source code. Recall is measured with respect to the known true clones. In most cases, the reference corpus is not complete – it does not contain every true clone within the subject system. In other words, the reference corpus is a proper subset of an oracle. Specifically, $\mathbf{R} \subset \mathbf{T}$. The goal in creating a reference corpus (i.e., a clone benchmark) is to efficiently build a large and varied reference corpus for accurate estimation of recall with a minimum of bias or subjectivity in the validation of the clones.

Recall is then measured as the ratio of the true clone pairs in the reference corpus that a clone detection tool is able to detect [13]. This is shown in Eq. 2.4, where **D** is the set of detected clone pairs detected by a tool, **R** is the known true clone pairs in the reference corpus, and **T** is the set of all true clones in the subject system(s). Given a sufficiently large and sufficiently varied reference corpus, we can assume that **R** approximates **T** for the purpose of measuring recall, even if $|\mathbf{R}|$ is much smaller than $|\mathbf{T}|$.

$$recall = \frac{|\mathbf{D} \cap \mathbf{T}|}{|\mathbf{T}|} \approx \frac{|\mathbf{D} \cap \mathbf{R}|}{|\mathbf{R}|} \qquad \mathbf{R} \subset \mathbf{T} \tag{2.4}$$

2.2.4 Clone Matching Algorithm/Metric

The measurement of recall with a reference corpus requires computing the intersection of the detected clones with the known true clones in the reference corpus. This requires determining which of the reference clones in the reference corpus are matched by the detected clones in the subject tool's clone detection report. However, clone detection tools do not always report clones perfectly, and may have some errors in the clone line boundaries. In particular, off-by-one line errors are common. Therefore, it is not as simple as searching the detected clones for an exact match of a reference clone. Additional sources of differences in clone boundaries is differing clone reporting styles between the reference corpus and clone detection tool, or even subjectivity in the precise boundaries of the reference clones themselves.

To accommodate these potential discrepancies, a clone matching algorithm/metric is used to evaluate if a given detected clone is a sufficient match of a specific reference clone for the reference clone to be considered as detected by the subject clone detector. This matching metric is evaluated for every detected clone until a sufficient match is found, or all detected clones are checked. This can computed efficiently if the detection report is inserted into an indexed database table and the algorithm is implemented as a database query.

Various clone matching algorithms have been used, that require the detected clone to: exactly match the reference clone (with or without an allowed line error), subsume the reference clone, intersect a specified ratio of the reference clone, and so on. The appropriate matching algorithm usually depends on the properties of the reference corpus and how it was constructed.

2.2.5 Methods for building a Reference Corpus

Many methods for building a reference corpus have been proposed or attempted, all with different advantages, weaknesses and challenges.

Manual Inspection One method is an exhaustive manual search of a subject system. Every pair of code fragments can be checked to see if they are a true or false clone, and added to the reference corpus. This is a reasonable approach for only very small subject systems. However, even a small system such as `cook` (51KLOC, 1244 functions) contains nearly one million pairs of code fragments at the function granularity alone [137], which is far too many candidate clones to check. One solution is to investigate only a statistically significant random sample of the code-fragment pairs [77]. However, since the chance that two randomly selected code fragments being a true clone is low, this is an inefficient way to build reference corpus.

Using Clone Detectors A common method is to build the reference corpus using the clone detection tools themselves.

The union method [13] builds a reference corpus as the union of the clones detected by a set of diverse clone detection tools. The flaw in this approach is it makes the unreasonable assumption that the tools have perfect precision. The false positives added to the corpus will harm the measure of recall. Additionally, the true clones not found by any of the tools are missing from the corpus. The code fragment pairs not found in the union cannot be assumed to be false clones, as it is not reasonable to assume the union of the tools has perfect recall for the subject system, so this corpus cannot be used to measure precision. An additional challenge is the handling of clones that are detected by multiple tools but are reported differently, including different reporting styles and precise line boundaries.

The intersection method [13] builds a reference corpus as the intersection of the clones reported by a set of diverse clone detection tools. The flaw in this method is it will trivially measure perfect recall for the tools used to build it. It cannot be assumed that the clones not found by all of the tools are false clones, so it can't be used to measure the precision of the tools either. It could possibly be used to measure the recall of non-participant tools, but the corpus will likely only contain the clones that are easy to detect (and thus detected by all the tools).

Another method is to combine these approaches [13], by taking the union, but removing those clones that are not detected by at least n tools, where $n > 1$. The idea is that those clones detected by more than one tools are less likely to be false positives. However, there is no clear choice of an appropriate value of n. Additionally, it can be the case that n tools report the same false positive, or that $n - 1$ tools detect a true clone. The resulting corpus may still be of low quality.

A popular approach is to combine the union method with statistical sampling and manual clone validation [13]. A random sample is selected from the clones reported by each clone detector and manually validated to identify the true and false clones. The sample size and distribution must be justified to be

statistically significant and fair amongst the tools. The resulting corpora can be used to estimate recall, while the validation efforts can be used to measure the precision of the participating tools. The resulting reference corpus is still biased by the detection capabilities of the participating tools, but may be sufficient to measure relative performance between the participants.

Search Heuristics An alternative to using the clone detectors themselves is to use search heuristics that are distinct from the clone detectors. Ideally, the search heuristic would be designed to have high recall at the cost of poor precision, with the true clones identified by manual inspection before inclusion in the reference corpus. This is similar to the manual inspection approach, except the search heuristic is used to greatly reduce the manual search space for better efficiency. However, heuristics could also cause some true clones to be missed and not included in the reference corpus. The heuristics may be designed to build a corpus with a particular context. Our implementation of this approach uses keywords and source-code patterns to identify code fragments implementing specific functionalities, which revealed large semantic clone classes after manual inspection [125]. Another approach uses Levenshtein distance to identify true clones as those meeting a specified threshold [80], without the need for manual inspection.

Clone Injection A reference corpus can be built by injecting known clones into a subject system, or authoring new clones within that software system. This is an alternative to mining for clones that already exist within the subject system. The advantage of this approach is it gives the benchmark creator total control over the clones in their reference corpus. However, manually creating interesting clones and injecting them into a software system is very effort intensive [13]. Perhaps only a small reference corpus can be built. However, the benchmark creator could carefully introduce interesting features into each clone and evaluate how this affects their detection by the subject clone detectors.

Artificial Clone Synthesis A reference corpus can be automatically synthesized by programmatically mimicking the creation of a clone by a software developer. We have done this using source-code mutation operators that mimic the types of edits developers make on copy and pasted code [107,111,131]. This is similar to manual clone injection, except the clones are constructed automatically. The advantage of this technique is a large corpus can be constructed with custom distribution of clone types. However, it is challenging to automatically synthesize complex and realistic clones, which is why benchmarks of real (developer-created) clones are also needed.

2.2.6 Measuring Precision

Compared to recall, it is easier to measure precision without an oracle. The precision of a clone detection tool (for a given subject software system) can be estimated by manually validating a statistically significant sample of its detected clones. Precision is then the ratio of the validated clones that are judged as true clones and not false positives. The precision of a clone detection tool can vary between software systems, so typically

this is repeated for a collection of diverse software systems from a variety of programming domains, and the precision measurement is averaged. While this procedure is rather simple, there are still some challenges. Clone validation is a very effort intensive process, and validating detected clones in a variety of software systems can take a significant amount of time. Clone validation is also subjective [9, 19, 21], so precision measured by different individuals could vary significantly.

Part I

Synthetic Clone Benchmarking with Mutation Analysis

In this part of the thesis, we present our work with synthetic clone benchmarking. We introduce the Mutation and Injection Framework and use it in a number of tool comparison studies. The Mutation Framework evaluates clone detection recall using synthetic clones in a mutation-analysis procedure. Synthetic benchmarking is needed to evaluate clone detection recall at a fine granularity for the different kinds of clones that can exist. Another advantage of synthetic benchmarking is it allows controlled recall experiments to be conducted, reducing or removing biases in the results. Fine-grained and controlled recall measurement is more difficult with real-world benchmarks, but real-world benchmarks evaluate for complex and realistic (developer-produced) clones, which is why both synthetic and real-world benchmarks are needed. In Part II we discuss our real-world clone benchmark, BigCloneBench.

The Mutation and Injection Framework procedure was previously proposed in the related work [107] and prototyped [111] for a single clone detection tool (NiCad). For this thesis, we improved the framework in the following ways: (1) we generalized the framework for compatibility with most clone detection tools, (2) improved the mutation operators and mutation process for better accuracy and control, (3) designed an evaluation procedure to allow recall to be compared across the clone types and clone edit types without bias, and (4) implemented the framework as an extensible tool. The framework enables the users to perform custom and fully automated recall evaluation experiments, which can then be shared, examined, repeated and extended by the community. We discuss the methodology and design of the Mutation Framework in Chapter 3.

In Chapter 4, we use our Mutation Framework to evaluate the state of the art clone detection tools per clone type. We compare our measurements against our expectations for the tools, and against the previous and popular clone benchmark: Bellon's Benchmark [13]. In this experiment, we validate the accuracy of the Mutation and Injection Framework, and demonstrate the need for synthetic benchmarking. We also show that Bellon's Benchmark may not be accurate for modern clone detection tools, creating the need for a new real-world clone benchmark, which is a motivation for our BigCloneBench (Part II).

In Chapter 5, we evaluate and compare the recall of state of the art tools at a fine granularity using our Mutation and Injection Framework. Specifically we measure the recall of the tools per edit type from the editing taxonomy for block and function granularity clones in Java, C and C# systems. In this study we demonstrate the advantage of the Mutation Framework's synthetic approach in evaluating the capabilities of the tools and pin-pointing their individual strengths and weaknesses.

In Chapter 6, we demonstrate how the Mutation Framework can be extended with custom mutation operators to evaluate clone detection tools for any kind of clone. In our case study, we synthesize Type-3 clones with a single dissimilar gap of variable length. We evaluate the robustness of the Type-3 clone detectors against Type-3 clones with various sizes of a gap. We generate a reference corpora that can evaluate the robustness of clone detection tools to small and large dissimilar gaps in otherwise identical Type-3 clones. We find that even the best of the state of the art tools struggle to detect Type-3 clones with a single dissimilar gap that is longer than three to five statements.

In Chapter 7, we adapt the Mutation Framework technologies to create ForkSim – a framework for generating datasets of artificial software variants (i.e., software forks) with known similarities and differences. These datasets can be used to evaluate tools for software variant analysis, such as for migrating variants towards a software product line architecture. ForkSim demonstrates how our mutation analysis technology can be used to benchmark clone detection and other software analysis tools, for various applications. We demonstrate the use of ForkSim by evaluating a clone detection tool for software variant analysis.

Chapter 3

The Mutation and Injection Framework

In this chapter, we present the Mutation and Injection Framework, a synthetic clone benchmarking framework that precisely evaluates clone detection recall at a fine granularity using a mutation-analysis procedure. The framework begins by selecting a random code fragment from a large repository of sample source code. It duplicates and mutates this code fragment to produce a code clone of a known clone type and with a known difference. The mutation operators used in clone synthesis are based on a comprehensive and empirically validated taxonomy of the types of edits developers make on copy and pasted code. The clone is then injected into a software system, evolving the system by a single copy-paste and modify clone. The clone detection tool is then executed for this software system and recall is measured for the injected clone. Since the framework created the clone itself, it is able to precisely evaluate the tool's detection of the clone, including if it appropriately handled the clone-type specific differences between the cloned code. This is repeated many thousands of times across all of the edit types in the taxonomy, allowing a comprehensive and exhaustive measurement of recall. The framework fully automates the recall experiment, and allows all aspects of the experiment to be customized and controlled.

We created the Mutation Framework to overcome challenges in Bellon's Benchmark [13], which has been the standard benchmark in clone detection for many years. Bellon built his benchmark by manually validating 2% of the clones detected by six contemporary (2002) tools for eight subject systems, requiring 77 hours of manual clone validation efforts. While the union may provide good relative performance evaluation between participating tools [13], there is no guarantee that subject tools have collectively detected all clones within the subject systems and therefore the measure of absolute performance is questionable. The reference corpus is therefore biased by the types of clones the participating tools detect. Baker [9] raised concerns with problems in the creation of Bellon's benchmark, including clone validation procedures. Charpentier et al. [19] revalidated a number of the clones and found disagreement in the results. The Mutation Framework overcomes these challenges by synthesizing clone benchmarks that are independent of the clone detection themselves, and which requires no subjective manual validation.

The Mutation and Injection Framework has some distinct advantages in measuring recall. It supports three programming languages (Java, C and C#) and two clone granularities (function and block). These are abstracted from the procedure, and the framework could be extended to additional languages and granularities. It is fully automated, and requires no manual clone validation efforts from the user. The framework

includes mutation operators for every type of edit developers make on copy and pasted code. This allows recall to be comprehensively measured at a finer granularity than clone type, allowing a tool's specific capabilities to be measured. The user configures the properties of the clones to be included in the synthesized reference corpus, including clone size, syntactical similarity, mutations and granularity. The user can therefore create a custom benchmark corpus for any general or specific cloning context to evaluate their tool against. Recall experiments produced by the framework can be easily replicated, duplicated, shared, modified and extended.

This chapter is based on a (currently unpublished) manuscript entitled "The Mutation and Injection Framework" and authored by myself and Chanchal K. Roy. The manuscript has been edited and reformatted to better fit this thesis.

This chapter is organized as follows. We discuss essential background knowledge in Section 3.1. We describe the framework's methodology in Section 3.2, and its usage in Section 3.3. We discuss the related work in Section 3.5, and conclude this work in Section 3.6.

3.1 Background

In this section, we provide additional background knowledge for this chapter. General background on clones and clone detection benchmarking can be found in Chapter 2. Here we describe the clone similarity metric used in this chapter and by the Mutation Framework. We also describe the editing taxonomy for cloning, which is an essential to our framework's clone synthesis process.

3.1.1 Clone Similarity

Clone similarity is the measure of the syntactical similarity between a clone pair's code fragments. It is expressed as a ratio between 0.0 (totally different syntax) and 1.0 (identical syntax). It can be measured by line, by statement, or by language token. The compliment of clone similarity is clone difference, the measure of the syntactical difference between a clone pair's code fragments. For this paper, and with our Mutation and Injection Framework, we measure clone similarity and difference using each code fragment's unique percentage of items (UPI).

Code fragments can be considered as either sequences of source code lines or language tokens. We can detect the differences in these sequences, from an editing perspective, using the Unix diff program (greatest common subsequences). A code fragment's UPI, with respect to another code fragment, is the ratio of its source code lines or tokens that are not found in the other code fragment, when also considering their order. In other words, the lines/tokens in the code fragment not matched to a line/token in the other code fragment by the greatest common subsequences algorithm. The UPI of code fragment f_1 with respect to code fragment f_2 is expressed mathematically in Equation 3.1, where *items* can be either source code lines or tokens. We measure clone dissimilarity as the larger of the clone's code fragment UPI, as shown in Equation 3.2, with clone similarity as its compliment, as shown in Equation 3.3.

Fragment#1	**Fragment#2**
void somefunction(int n) {	void somefunction(int n) {
int sum = 0;	int sum = 0;
int product = 1;	
for(int i = 0; i < n; i++) {	for(int i = 0; i < n; i++) {
sum = sum + i;	sum = sum + i;
product = product * i;	
}	}
	for(int i = 0; i < n; i++) {
	product = product * i;
	}
return new SomeClass(sum, product);	return new SomeClass(sum, product);
}	}

#Unique Lines = 2
#Lines = 9
UPI = 0.22
	#Unique Lines = 3
	#Lines = 10
	UPI = 0.30

Difference: 0.3, Similarity: 0.6	

Table 3.1: Clone Difference and Similarity Example

$$upi(f_1, f_2) = \frac{\#\ unique\ items\ in\ f_1\ by\ diff(f_1, f_2)}{\#\ of\ items\ in\ f_1} \tag{3.1}$$

$$dissimilarity(f_1, f_2) = max(upi(f_1, f_2), upi(f_2, f_1)) \tag{3.2}$$

$$similarity(f_1, f_2) = 1 - dissimilarity(f_1, f_2) \tag{3.3}$$

An example for line items is shown in Table 3.1. This table shows the item matching as done by the diff algorithm. Fragment#1 is 9 lines long and has 1 unique line, a UPI of 22%. Fragment#2 is 10 lines long and has 3 unique lines, a UPI of 30%. The difference for this clone is the larger of the UPI values, 30%. Similarity is the compliment of difference, 60%. The token metric is evaluated in an identical fashion. By inserting a newline between each token, the line-based Unix diff algorithm can be used to evaluate the token based similarity metric without modification.

Cloned code fragments often contain Type-1 (white-space, formatting and layout) as well as Type-2 (identifier name and literal value) differences. These differences can greatly lower our clone similarity measurement. These types of differences are often ignored in the cloning context, and a more accurate clone similarity is measured if these differences are normalized. We therefore apply Type-1 and Type-2 normalizations to the code fragments before measuring clone similarity. Our Type-1 normalization applies a strict pretty-printing, which results in a single statement per line with normalized whitespace, and removes all comments and blank lines. Our Type-2 normalization replaces each identifier with 'X', and each literal with '0'. With these normalizations, all Type-1 and Type-2 clones will have a similarity of 100%, while Type-3 clones will have

a similarity less than 100%. The clone difference will therefore measure the amount of Type-3 differences between the code fragments.

3.1.2 The Editing Taxonomy for Cloning

The basis of our clone synthesis technique is the ability to simulate a developer's copy, paste and modify cloning behavior. The foundation of the modification step is a comprehensive taxonomy of the types of edits developers make on copy and pasted code fragments. This taxonomy was created by Roy et al. [108] and was constructed based upon a literature survey of clone types, clone taxonomies, and empirical studies. The taxonomy consists of fifteen editing activities that produce clones of the first four clone types. The taxonomy was empirically validated against copy, paste and modify cloning patterns observed in clones from seventeen open source Java and C systems. They are confident that their taxonomy is capable of modeling all clone types defined in the literature. The editing taxonomy is not specific to any particular development task, such as software maintenance.

The taxonomy is described in Table 3.2, including a description of each editing activity and the clone type they produce. This is a slightly modified version of the original taxonomy, reducing the 15 editing activities to 14 due to changes in interpretations of the clone types. Specifically, we removed the Type-2 editing activity: "Replacement of identifiers with expressions (systematically or non-systematically)" [108]. In previous benchmarks this type of edit was considered a Type-2 clone [9, 13]. However, the modern clone detection tools typically interpret this change as Type-3, and target their detection using similarity thresholds rather than normalizations. This activity can be safely removed as the existing Type-3 editing activities already cover it, specifically editing activities 7 and 8. Further details of the editing taxonomy are available in its original publication [108].

An example of each editing activity is shown in Figure 3.1. We begin with an initial code fragment at the top left position. We then sequentially apply an example of each editing activity onto the code fragment in numerical order. The number on the arrows indicate the editing activity applied, and point from the code fragment before the edit is applied and to the code fragment after the activity is applied. Each version of the code fragment highlights the changes made by the most recent editing activity. The final version at the bottom left is after an example of all 14 editing activities having been applied to the original code fragment.

3.2 The Mutation and Injection Framework

The Mutation and Injection Framework measures the recall of clone detection tools using a mutation analysis procedure. It is a fully automatic framework that requires no manual efforts during either the construction of the reference corpus nor the evaluation of the subject tools. It achieves this by synthesizing a reference corpus of artificial clones using source-code mutation and injection, rather than mining for real clones in a subject system. An advantage of the framework is that it requires no manual clone validation, which has

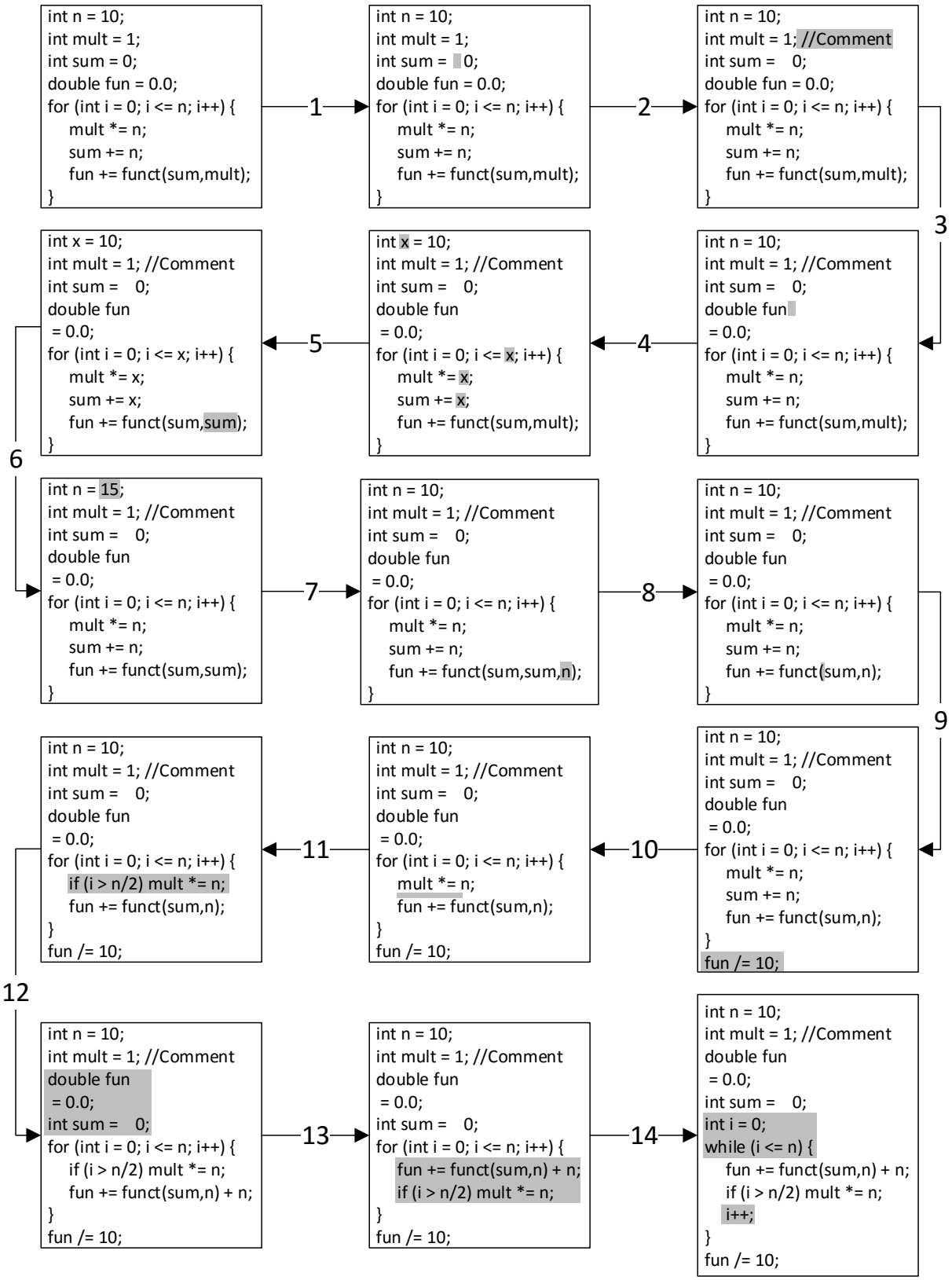


Figure 3.1: Clone Synthesis Example

Table 3.2: Editing Taxonomy for Cloning

ID	Edit Description	Clone Type
1	Change in whitespace.	
2	Change in commenting.	Type-1
3	Change in formatting.	
4	Systematic renaming of an identifier.	
5	Renaming of a single identifier instance.	Type-2
6	Change of a literal value.	
7	Small insertion within a line.	
8	Small deletion within a line.	
9	Insertion of a line.	Type-3
10	Deletion of a line.	
11	Modification of a line.	
12	Reordering of declaration statements.	
13	Reordering of statements.	Type-4
14	Replacement of one type of control statement with another.	

been an obstacle in measuring recall. By synthesizing the reference corpus, its properties can be controlled, and potential biases can be avoided. The framework evaluates tools using the following procedure, which is also shown in Figure 3.2.

1. A single clone pair is added to a software system by simulating the creation of a copy, paste and modify clone by a developer. This is accomplished by duplicating and mutating a source code fragment using cloning mutation operators, and introducing this new clone pair into a subject software system using source-code injection.

2. The subject clone detection tool is executed for this mutant version of the subject system.

3. The tool's unit recall is measured specifically for the detection of the injected clone.

4. Steps 1-3 are repeated for a large number and variety of clone pairs.

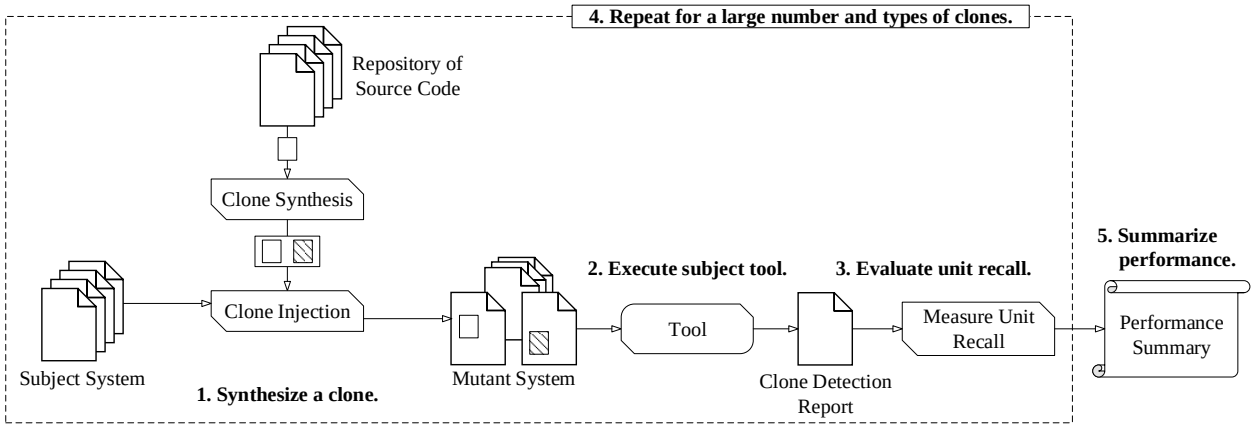


Figure 3.2: Overview of the Mutation Framework Procedure

5. The tool's average recall across these mutant systems (the reference corpus) is reported.

This is a canonical mutation analysis procedure, very similar to that used in mutation testing. In mutation testing, mutation operators are used to randomly introduce a bug into a software system and the system's testing strategy is evaluated for its ability to detect and isolate the synthetic bug. To evaluate clone detection tools, we inject duplicate code into a system, and use mutation operators to introduce a random difference between the duplicated code fragments corresponding to one of the clone types. The subject clone detection tool is evaluated for its ability to detect the synthetic clone. The mutation operators are based upon a comprehensive taxonomy of the types of edits developers make on cloned code (Section 3.1.2). This strategy allows us to measure the subject tool's recall not only per clone type but also per clone edit (mutation) type, which provides greater insight into the performance, capabilities and weaknesses of the particular clone detection tool.

The implementation of the framework splits the mutation analysis into two discrete phases: the generation phase and the evaluation phase. During the generation phase, the clones are synthesized and injected into distinct copies of a subject system. These mutant systems and the injected clones they contain form the reference corpus. The framework allows many constraints to be placed on the generated corpus in order to control its properties. The evaluation phase executes the subject tools for the mutant systems, and measures their recall for the injected clones. The implementation is split in this way to allow the corpus to be generated ahead of time and then shared and reused in multiple tool evaluation experiments.

In the following subsections we describe the framework in full detail. We begin by describing how the clones are synthesized. Next we outline the generation phase, during which clone synthesis is used to create the reference corpus. Then we outline the evaluation phase, and how it automates the subject tool execution and evaluation for the reference corpus.

3.2.1 Clone Synthesis

The framework synthesizes clone pairs by mutating real code fragments mined from a source code repository. The mutations are based upon the editing taxonomy for cloning (Section 3.1.2), which is a validated and comprehensive taxonomy of the types of edits developers make on copy and pasted code. This allows a comprehensive corpus of realistic clone pairs to be synthesized. Mutations are applied by *mutation operators* that take a code fragment as input and output the same code fragment with a single random modification of their edit type. The framework user specifies the types of clones to be synthesized using *mutators*, which are sequences of one or more mutation operators. The mutator applies the mutation operators one by one, in order, to an input code fragment. The original code fragment produced by a real developer, and its mutant code fragment produced by a mutator, form a synthetic clone pair produced by mimicking the copy, paste and modify cloning activities of a real software developer. The framework currently supports the synthesis of Java, C and C# clone pairs of the first three clone types at the function and block syntax granularities, although it could be extended to other language.

An example of clone synthesis is shown in Figure 3.3. The original code fragment is mutated by a mutator with a mutation operator sequence of length three. The mutation operators apply a single random change of their edit type. The first mutation operator changes formatting, the second changes the value of a literal, and the third adds a comment. The mutation operators are applied, in this order, to a copy of the original code fragment. The mutations have been highlighted in the final mutant code fragment. The first and third mutation operator apply Type-1 clone differences, while the second mutation operator applies a Type-2 clone difference. Therefore the original and mutant code fragments form a Type-2 clone pair.

In the following sections, we describe the mutation operators and mutators in more detail.

Mutation Operators

The mutations are performed by *mutation operators*, a concept of mutation analysis. From the editing taxonomy, we created fifteen mutation operators that mimic the types of edits developers make on copy and pasted code. These mutation operators take a code fragment as input, and output the same code fragment with a single random edit of the operator's defined edit type. Our mutation operators are summarized in Table 3.3. This table lists each mutation operators' name, edit type, how the edit type is realized in its implementation, and the clone type the edit belongs to.

Our mutation operators cover the types of edits that produce clones of the first three clone types. We did not create mutators for the Type-4 editing activities as very few tools are able to handle them. The cost of adding them outweighed the benefit of including them, so we leave them to future work when more clone detection tools consider Type-4 clones. In the interest of evaluating clone detection at as fine of a granularity as possible, some of the editing activities from the editing taxonomy were split into multiple mutation operators.

The Type-1 and Type-2 mutation operators are able to perform any modification of their edit type that is possible for an input code fragment. For example, the mCW_A operator can add whitespace at any syntactically valid location in the input code fragment. The Type-3 mutation operators instead perform an example of their type of edit. It is not reasonable to implement the Type-3 operators to be able to apply any valid modification of their type to an input fragment. For example, for the "injection of a line" mutation operator, there is an infinite number of possible lines of syntax that could be inserted. Selecting a line from the fragment itself and duplicating it within the fragment at a randomly selected line is a sufficient approximation from the point of view of a clone detection tool.

The mutation operators are implemented in TXL [27], a source transformation language. They use a simple language-dependent grammar to parse the input code fragment into a syntax tree. The grammar captures both the syntax tokens and the white space (formatting) of the input fragment. The operator's mutation is implemented using a subtree search and replacement pattern. The operator applies the mutation by replacing exactly one randomly selected subtree that matches the search pattern with the corresponding replacement pattern. The output fragment is produced by un-parsing the modified syntax tree. The result

Table 3.3: Mutation Operators

Name	Edit	Implementation	**Clone Type**
mCW_A	Addition of whitespace.	A tab or space character is inserted between two randomly chosen tokens.	Type-1
mCW_R	Removal of whitespace.	A (syntactically redundant) random tab or space character is removed.	Type-1
mCC_BT	Change in between token (/* */) comments.	A /* */ comment is added between two randomly chosen tokens.	Type-1
mCC_EOL	Change in end of line (//) comments.	A // comment is added at the end of a randomly chosen line.	Type-1
mCF_A	Change in formatting (addition of a newline).	A newline is inserted between two randomly chosen tokens.	Type-1
mCF_R	Change in formatting (removal of a newline).	A randomly chosen newline is removed.	Type-1
mSRI	Systematic renaming of an identifier.	A randomly chosen identifier, and all of its occurrences, are renamed.	Type-2
mARI	Arbitrary renaming of an identifier.	A randomly chosen single instance of an identifier is renamed.	Type-2
mRL_N	Change in value of a numeric literal.	The value of a randomly chosen numerical literal is changed.	Type-2
mRL_S	Change in value of a string or character literal.	The value of a randomly chosen string or character literal is changed.	Type-2
mSIL	Small insertion within a line.	A parameter is added to a randomly chosen method call or signature.	Type-3
mSDL	Small deletion within a line.	A parameter is deleted from a randomly chosen method call or signature.	Type-3
mIL	Insertion of a line.	A line of source code (containing a single statement) is inserted at a random line in the code fragment.	Type-3
mDL	Deletion of a line.	A randomly selected line (containing a whole single-line statement) is deleted.	Type-3
mML	Modification of a whole line.	A randomly selected line is modified by placing it in a single-line if statement. For example 'x = 15*y;' becomes 'if(X==Y) x=15*y;'	Type-3

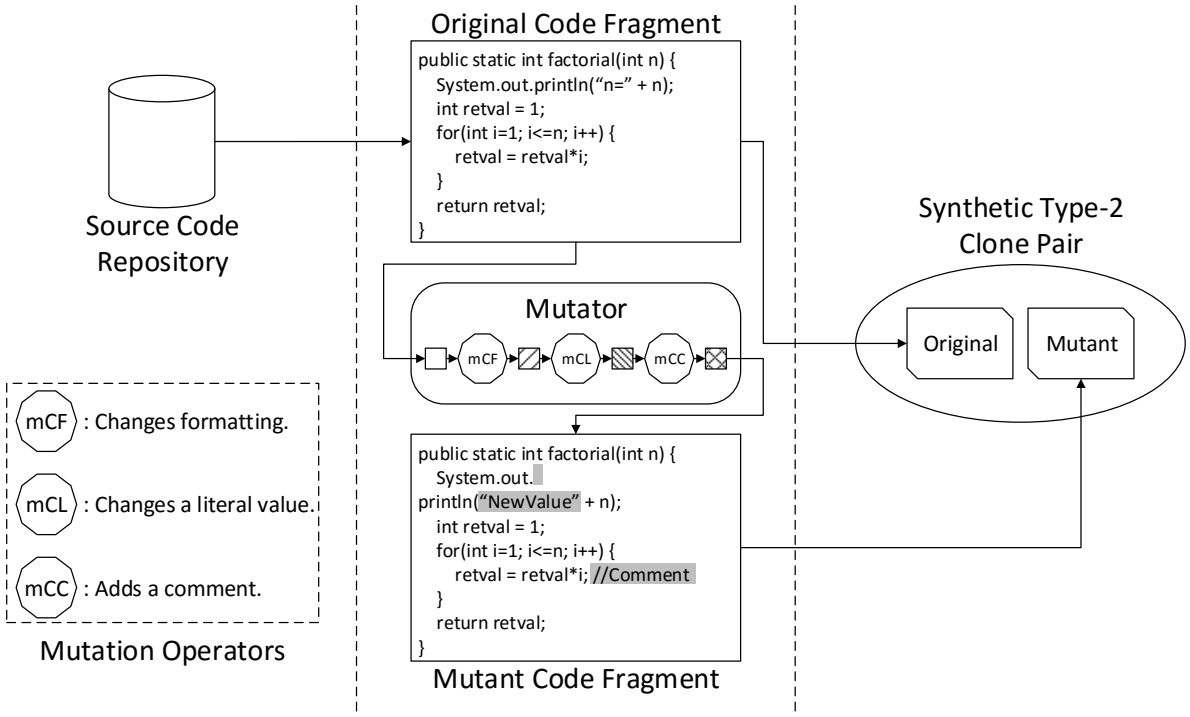


Figure 3.3: Clone Synthesis Example

is an output code fragment that differs from the input code fragment by a single random application of the intended mutation. The input fragment's syntax and formatting is otherwise unmodified in the output fragment. The operators are able to detect when they can not be applied to a particular input fragment (i.e., when there is no matches to the search and replace pattern), in which case they return an error.

The operators exist as independent programs that are registered with the Mutation Framework. While our mutation operators are registered by default, the user is free to register their own custom mutation operators implemented in any technology of their choosing. Our mutation operators target general clone detection, and cover our clone taxonomy for the first three clone types. Users of the framework might design their own mutation operators to target specific clone detection use-cases, or for specific targeted debugging of their tool (we show an example of this in Chapter 6). Custom mutation operators must conform to the same input/output procedure. The mutation operator receives the code fragment and language as input, and outputs the mutated code fragment.

The framework includes mutation operators implemented for Java, C and C# source code. Their implementations rely on simple token grammars for these languages. The mutation operators can be easily adapted to support additional languages. In particular, for C-style languages like Java, C and C#, the mutation operators can be adapted to a new language by simply adding the language's token-based grammar, with no changes the mutation search/replace patterns.

Mutators

The framework user specifies the kinds of clones to be synthesized for the reference corpus using *mutators*, which are sequences of one or more mutation operators. A mutator synthesizes a clone pair by executing its mutation operators as an input/output chain, supplying the input code fragment to the first operator, and retrieving the mutant code fragment from the last mutation operator. Any number of mutators may be defined using the default and custom mutation operators. By specifying the set of mutators used to synthesize clones, the framework user can create a reference corpus tailored for the cloning context they wish to evaluate their subject clone detection tools for.

Since the mutation operators are implementing using a simple grammar they can, in rare cases, introduce a syntax error. To prevent these errors from contaminating the reference corpus, the mutator validates the syntax of the mutant code fragment after each application of a mutation operator using a full language grammar. It also checks that the change applied corresponds to the clone type of the mutation operator. If a problem is detected, the edit is discarded and the mutation operator is re-attempted. A user-defined number of mutation operator attempts are tried before the mutator returns an error. If one of the mutation operators cannot be applied to the fragment, then the mutator returns an error immediately. These checks guarantee the integrity of the synthesized clone pairs, and therefore the integrity of the reference corpus.

The mutators also support a number of constraints to be placed on the synthesized clones, including: clone size, clone similarity, and mutation containment. The clone size constraint allows the framework user to specify the minimum and maximum size, by line and by token, of a synthesized clone pair's original and mutant code fragments. The clone similarity constraint allows the framework user to specify the minimum clone similarity, measured by line or by token as in Section 3.1.1, of the synthesized clones. These constraints can be used to shape the context and properties of the reference corpus. Clone size and similarity are common clone detection tool configurations. These constraints allow the subject tools to be appropriately configured to target the benchmark. Otherwise, the tools may be configured poorly for the benchmark, causing bias in the measured recall.

The mutation containment constraint guarantees that the mutations occur a particular line distance from the start and end of the original code fragment. Containment is specified as a percentage of the original code fragment's size. For example, with a mutation containment of 20%, if a mutator is given a 10 line code fragment, it will only allow mutations on lines 3 through 8. The first and last 20% (lines 1-2 and 9-10, respectively) of the code fragment are left unaltered. This constraint ensures that the code fragments of a synthesized clone begin and end with exactly cloned code. If the mutation occurs on or very near the edge of the original code fragment, then the mutation may actually be considered external to the clone pair. We want the mutation to be a part of the clone so we can measure how well the clone detection tools handle these particular kinds of clone differences. The mutation containment constraint ensures the introduced clone differences occur deep enough within the code fragments for them to be, without a doubt, an essential part of the clone pair. For a clone detection tool to successfully detect this clone, it must therefore include the

clone differences as part of its detection of the clone.

It is difficult to program the mutation operators to respect these constraints. Instead, the mutator checks the containment constraint after each application of a mutation operator, and the clone size and minimum similarity constraints after the application of the mutator's mutation operator list. If a constraint is violated, the mutator will reattempt a mutation operator (containment) or the entire mutation operator list (size/similarity) until a satisfactory clone pair is produced, or a user-specified maximum attempt threshold is reached. If it fails to meet the constraints, the mutator will return an error. While this increases the clone synthesis time somewhat, it simplifies the implementation of the mutation operators, which need to be error-free.

The framework defaults and recommends the use of a single-operator mutator for each of the registered mutation operators. This creates a reference corpus that measures a tool's performance at the edit type granularity. Multi-operator mutators allow a reference corpus with more complex clones to be generated. For example, a mutator set could be defined for a variety of sequences of mutation operators that produce Type-1 clones. This would allow strong evaluation of tools specifically for a variety of Type-1 clones. However, higher order mutations pose some risk. As more operators are applied, it becomes difficult to predict how the operators may interact. Later mutation operators in a mutator's sequence may even reverse previous ones. Additionally, when mutation operators are mixed, you lose the ability to measure performance per edit type. Higher-order mutations are advanced use of the framework, and require careful attention and interpretation. The default single-operator mutators are recommended for standard tool evaluation usage.

3.2.2 Generation Phase

During the generation phase, the reference corpus is constructed by synthesizing clones and injecting them into unique copies of a subject system. An overview of this phase is shown in Figure 3.4. The generation process begins by selecting a random code fragment from a repository of source code. This selected code fragment is then mutated by m user-defined clone-synthesizing mutators. The resulting m mutant code fragments differ from the selected code fragment by a random application of the mutation defined by their mutator. The mutant code fragments are paired with the selected code fragment to form m synthesized clone pairs. For each of these clone pairs, i mutant versions of the subject system are created by injecting the selected and mutant code fragments into the subject system at a random syntactically correct locations. Each of these mutant systems evolve the subject system by a single copy, paste and modify clone. In total, mi mutant systems are created from a single randomly selected code fragment. This process is repeated for n randomly selected code fragments in order to create a reference corpus containing nmi unique clone pairs. Each reference clone pair is contained within its own mutant version of the subject system. This is done so the subject tools may be evaluated for each of the reference clones in isolation.

A database is used to track the specification of each selected code fragment, mutant code fragment, and mutant system. The selected and mutant code fragment text is stored in files referenced by the database.

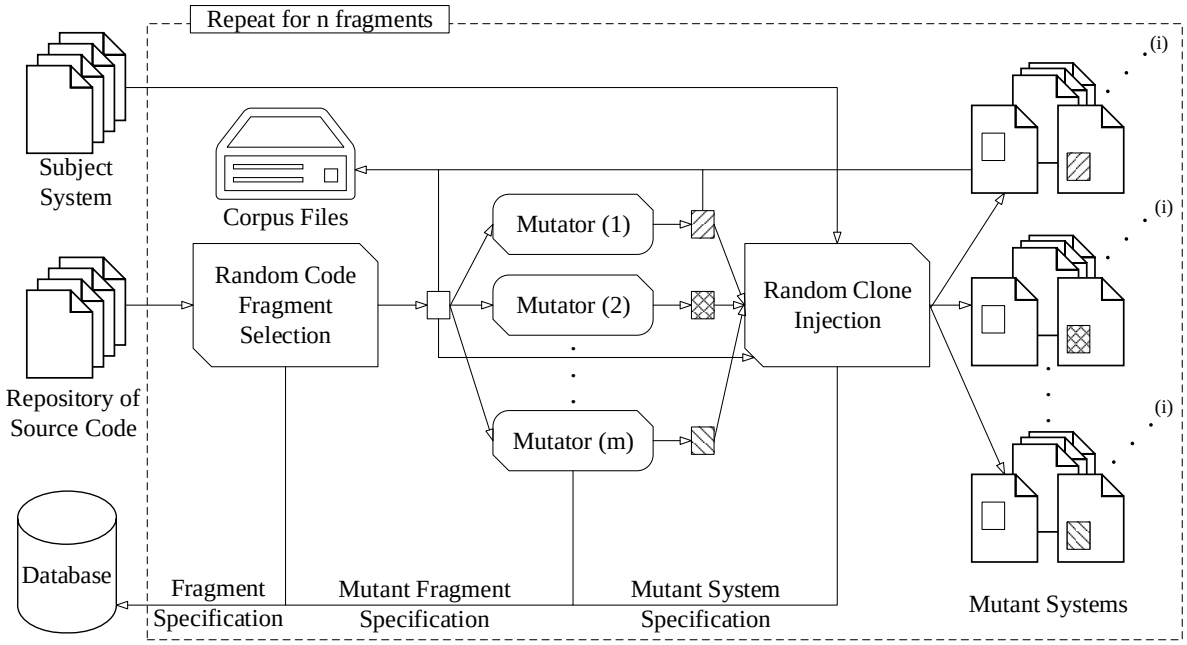


Figure 3.4: Overview of Generation Phase

This information can be used to construct any of the mutant systems, which contain a single clone from the reference corpus. Clone detection tools can be evaluated for the reference corpus by executing them for each of the mutant systems and evaluating their recall for the injected clone, which is the role of the evaluation phase.

In the following sub-sections we explore the generation phase in detail. We begin by discussing how the generation phase can be configured to control the properties and contents of the generated reference corpus. Next, we detail the steps of the generation phase as they are executed per selected code fragments. There are two primary steps: (1) selecting and mutating a selected code fragment, and (2) injecting the resulting clone pairs into copies of the subject system.

Configuration

The generation phase is highly configurable, which provides the user full control over the size, properties and contents of the generated reference corpus. The user provides a repository of source code from which real code fragments are selected for mutation. The user must also provide the subject software system the synthesized clones are injected into. The user specifies the types of clones to be generated for the corpus by defining the set of clone-synthesizing mutators (Section 3.2.1) to be used. The generation phase also allows numerous constraints to be placed on the generated corpus, such as the language and granularity of the clones, the size of the corpus, and various clone properties. In this section we describe these configurations and their effect on the generation phase.

Source Code Repository The user provides a source code repository from which real code fragments are selected for clone synthesis. The repository may be any collection of source code of the target programming language. Ideally, the repository should be large and varied. For example, a combination of the Java standard library and popular 3rd party libraries (e.g., Apache Commons) is a good source code repository for generating a Java clone reference corpus.

Subject Software System The user provides a subject software system into which the clones are injected. Any subject system of the target language will do, so long as its large enough to have a variety of injection locations. Since each clone in the reference corpus is injected into its own copy of the subject system, the subject clone detection tools have to be executed for mutants of this system many times. Therefore, the subject system should be small enough that the subject tools can be executed for it perhaps thousands of times within a reasonable time frame.

Mutator Set The user can specify any number of mutators by specifying the sequence of mutation operators they should apply to the selected code fragments. The framework's default mutator set includes 15 single-operator mutators, one for each of the 15 default mutation operators. The default mutators mutate the selected code fragment with a single instance of their assigned mutation operator. This creates a reference corpus that can measure the tool's performance at the edit type (mutation operator) granularity.

Generation Constraints The generation constraints allow the size, scope and clone properties of the generated reference corpus to be specified. The available constraints are as follows:

Clone Granularity The granularity of clones to synthesize. The framework supports clone synthesis at the function and block (a code segment defined by an opening and closing bracket, i.e., {...}) granularities.

Language The programming language of the generated reference corpus. The framework supports the synthesis of Java, C and C# clone pairs.

Number of Selected Code Fragments The maximum number of code fragments to select for clone synthesis.

Injection Number The number of times to inject each synthesized clone pair. In other words, the number of mutant systems to create per clone. Each injection of a clone uses a different injection location in the subject system.

Clone Size The minimum and maximum size of a clone's code fragments. Specified by line and by token, independently.

Minimum Clone Similarity Minimum clone similarity, measured by line and by token using the UPI method described in Section 3.1.1, after Type-1 and Type-2 normalization.

Mutation Containment The minimum distance the mutation must be from the edges (start/end) of the selected code fragment, specified as a fraction of the fragment's size in lines.

Fragment granularity and *language* are used to set the scope of the experiment. By limiting a corpus to a particular programming language and clone granularity, more specific performance evaluation can be accomplished. The intention is for the user to perform multiple experiments with the framework using the different permutations of clone granularity and language. Performance can then be individually measured per language and granularity.

The *number of selected code fragments*, n, the *injection number*, i, and the size of the mutator set, m, define the maximum size of the generated reference corpus: mni clone pairs. The framework will continue to select code fragments for clone generation until its specified maximum is reached, or all eligible fragments in the repository have been exhausted. A larger number of selected code fragments, n, results in larger diversity in the syntax of the reference corpus's clones. A larger number of injection locations per clone, i, increases the diversity in clone location in the reference corpus's clones. Both are essential for creating a reference corpus that accurately measures recall.

The *clone size* and *minimum clone similarity* constraints make it easier to configure the subject clone detection tools. Most tools are parameterized by clone size and clone similarity thresholds, which are used to limit the scope of their clone search and reporting processes. These parameters may also affect a tool's execution time and precision. By placing similar constraints on the reference corpus, the tools can be properly and confidently configured for the corpus. This reduces the incidence of inaccurate performance measurement due to configuration mismatch between the tool and reference corpus, which is a significant threat in benchmarking [139]. These constraints may also be used to constrain the corpus for evaluating specific cloning contexts. For example, a corpus could be generated for evaluating the detection of only small or large clones, or a corpus could be made for very similar or less similar clones.

The *mutation containment* constraint is used to ensure that the mutations are an integral part of the synthesized clone pairs. The framework's goal is to measure how well the tools perform for specific differences between cloned code fragments. If the mutation is too near the edge of the clone, it may actually be external to the clone. This constraint ensures that the mutation occurs far enough away from the edges of the clone's code fragments that it is a guaranteed component of the clone pair. And therefore it is correct to require a clone detection tool to handle the mutation to have successfully detected the clone. Mutation containment is specified as a ratio of the size of the selected fragment measured in lines. For a mutation containment of 20% and a selected fragment 10 lines long, all mutant fragments produced will not modify the selected fragment's first or last 2 lines.

The configurable mutators and generation options allow the framework user to generate a variety of well understood benchmark reference corpora. They allow the user to generate a corpus targeting the cloning context they wish to evaluate their tools for. By generating multiple corpora using different configurations, the user can evaluate their tools for a variety of cloning contexts. Having a corpus with exactly known

properties ensures that the performance results are correctly interpreted. It also allows the tools to be properly configured for the benchmark.

Step 1 - Code Fragment Selection and Mutation

The first step of the generation phase is to select a real code fragment from a source repository, and mutate it with the m user-defined mutators to produce a set of m synthetic clone pairs. The framework begins by extracting all of the code fragments in the source code repository that satisfy the clone language, granularity and size constraints. The code fragments are tracked by a data structure that allows random selection without repeats.

The data structure is queried for a random code fragment that has not been selected previously. This selected code fragment is mutated by each of the m user-defined mutators, and the output mutant fragments are collected. The mutators are configured to only produce mutant fragments that satisfy the clone size, minimum clone similarity, and mutation containment constraints. A mutator will produce an error if the mutation can not be applied, or if the constraints cannot be satisfied. If all of the mutators are successful, then the selected code fragment is paired with each of its mutant code fragments, and the resulting clone pairs are added to the reference corpus.

If at least one of the mutators fail, for any reason, then the selected fragment and its mutant fragments are discarded, and a new code fragment is selected. This is done to ensure that each mutator contributes the same number of clones to the reference corpus, and to ensure that their clones originate from the same set of selected code fragments. This makes it possible to measure and compare a subject clone detector's recall per mutator without bias due to the code fragment selection or number of synthesized clones per mutator.

This process is repeated until either the maximum number of selected code fragment constraint is reached (not counting the selected code fragments that are discarded), or when all of the code fragments in the repository are exhausted. For a large enough source repository, exhausting the available code fragments should not occur. The generated clones are tracked by a database, including: the origin and text of the selected code fragment, and the mutator and text of its mutant fragments. These clone pairs are now ready for injection into the subject system.

Step 2 - Clone Injection and Mutant Systems

For each of the synthesized clone pairs, one or more mutant systems are created by randomly injecting the clone into the subject system. The mutant system differs from the original subject system by a single copy, paste and modify clone. The injection simulates the development of a new code fragment in the system (injection of the selected code fragment of the clone pair) followed by the cloning and modification of this fragment (injection of the mutant code fragment of the clone pair).

Each clone pair is injected into its own unique copy of the subject system in order to minimize the amount of simulated development performed on the subject system, and to prevent the injected clones from

interacting. This allows the framework to evaluate the clone detection tools for each injected clone in isolation, and prevents the properties and structure of the mutant system from diverging too far from that of a real software system.

Clone injection location depends on the clone granularity. Function clones are injected by selecting two random functions in the subject system, and injecting the selected (function) code fragment after one of these functions, and the mutant (function) code fragment after the other. The clone is injected after existing functions rather than before in order to prevent the injection from separating an existing function from its in-code documentation (e.g., javadoc in Java), which is typically placed before the function. Block clones are injected by selecting two random code blocks from the subject system, and injecting the selected (block) code fragment within one of these blocks, and the mutant (block) code fragment within the other. For simplicity, block code fragments are injected either at the start or the end of the chosen code block. A code block can be safely injected at the start or end of an existing code block without creating a syntax error. To simplify tracking of the injected clone, and to prevent any one file in the subject system from diverging too far from its original state, the selected and mutant code fragments are always injected into different source files.

The injection process guarantees that the modified files remain syntactically correct after the injection of the clones. As the integrity of the generated reference corpus is very important, the framework verifies this by validating the modified source files against a full language grammar. While syntactically correct, the modified files may not compile. The injected code may refer to global variables, fields, functions, types, etc. that do not exist within the subject system. This is not a problem as most clone detectors do not require compiled code, or even compilable code, so long as the code is syntactically correct (i.e., can be parsed). Additionally, it is unlikely that the injected code will be semantically compatible with the source files it is injected into. As the framework focuses on syntactical clones and syntactical clone detectors, the semantic mismatch will not affect the tool evaluation.

The *injection number* configuration controls the number of mutant systems the framework creates per generated clone. Each additional mutant system created for a clone will use a different injection location. Using an injection number greater than one is important to the quality of the reference corpus. A clone detection tool may fail to detect an injected clone not due to the syntax of the clone or its clone difference, but due to its location. For example, the tool might be unable to parse the file(s) the clone was injected into due to limitations in its parser, or it may fail to search for clones contained in these files. In this case, the tool may have successfully detected the clone if it had been injected elsewhere. This way, a tool's average recall for different injections of the same clone better reflects it detection of that clone. By varying the injection location, the overall recall of a tool can be more accurately measured.

Injection locations are chosen per selected code fragment rather than per clone. Therefore, all clones originating from the same selected fragment use the same injection location (or set of injection locations if the injection number is greater than one). This way, if the reference corpus is split per mutator, each sub-corpus has the same injection location representation and variance. Recall can then be measured and

compared per mutator without bias due to the injection locations used.

In total, mni mutant systems are created, where m is the size of the mutator set, n is the number of selected code fragments, and i is the injection number. The number of mutant systems can be very large. It is not space efficient to store a copy of each of these mutant systems as they differ from the subject system by only the injected clone. The generation phase builds the mutant systems to verify their integrity, but deletes them afterwards. Instead it stores only the specifications of the mutant systems. A mutant system is specified by the clone to be injected (a selected code fragment and one of its mutant code fragments), and the injection location (the source files and file positions to inject at). The mutant systems are then constructed as needed during the evaluation phase using this specification. This keeps the reference corpus small enough for storage on a high performance drive as well as for convenient distribution over the Internet.

3.2.3 Evaluation Phase

During the evaluation phase, the subject clone detection tools are evaluated for the reference corpus synthesized during the generation phase. The subject tools are executed for each of the mutant systems, and their recall is measured specifically for the injected clones. Remember that the reference corpus contains nmi mutant systems. The n selected code fragments were mutated by m mutators to produce nm clone pairs that were each injected into i copies of the subject system to produce nmi mutant systems, each containing a single injected clone pair. The evaluation phase automates the execution of the subject clone detectors, requiring only a simple communication protocol be implemented for the tool. This protocol, implemented by a *tool runner*, allows the framework to configure and execute the tool, as well as collect and understand its clone detection report.

The evaluation phase is depicted in Figure 3.5. This processes is repeated per mutant system in the reference corpus. First, the mutant system is constructed from its specification in the reference corpus. Next, the subject clone detection tools are automatically executed for the mutant system using their tool runners. The framework collects and stores the resulting standardized clone detection reports. These reports are analyzed to measure each tool's unit recall specifically for the injected clone in the mutant system.

A unit recall of 1.0 is assigned to a subject tool for a mutant system if the tool successfully detected and reported the injected clone, otherwise it is a assigned a unit recall of 0.0. Successful detection is determined by a clone matching algorithm. This algorithm requires the tool to report a clone pair that: (1) subsumes the injected clone, within a given tolerance, (2) handles the clone-type defining mutation in the injected clone, and (3) exceeds a minimum clone similarity threshold. The reported clone is required to subsume the injected clone, rather than exactly match it, as there may be additional cloned code surrounding the injected clone due to the choice of injection location (at least in the case of block granularity clones). The reported clone must include the mutated portion of the injected clone, as the goal of this framework is to see how well the tools are able to handle these particular differences between cloned code fragments. The reported clone is required to exceed a given similarity threshold to prevent false positives reported by a tool, that happen to subsume

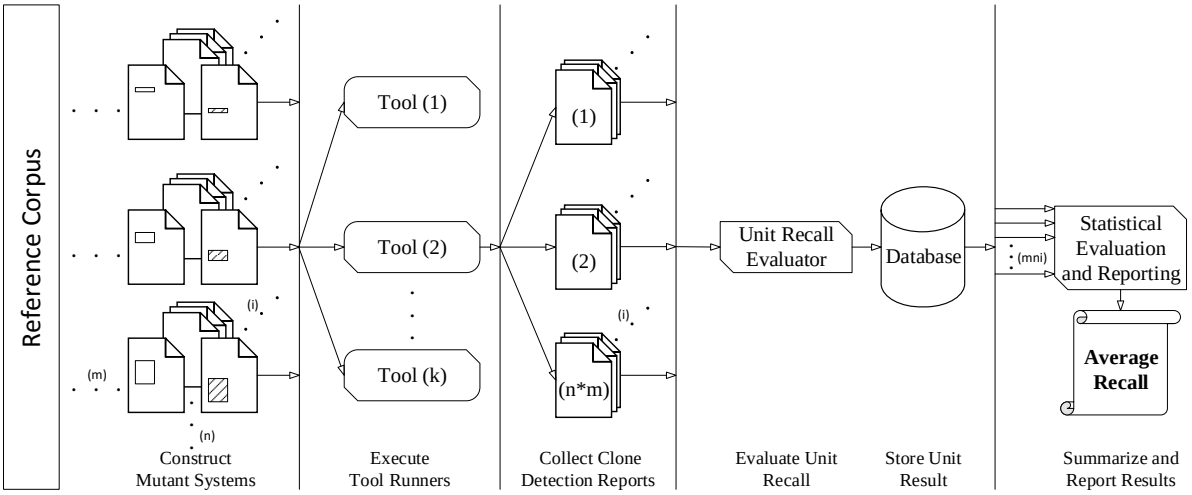


Figure 3.5: Overview of Evaluation Phase

the injected clone by chance, from being accepted as a successful match. Therefore, a successful match is a true clone pair that captures the injected clone and handles the particular clone differences introduced by mutation.

The framework measures the subject tools' unit recall for each of the mutant systems. It aggregates these results and reports the subject tools' average recall for various subsets of the reference corpus. Specifically, the framework reports the subject tools' recall per clone type, per mutator, and per mutation operator.

In the following subsections, we describe the evaluation phase in detail. We begin by overviewing its configurations options. We then describe the subsume-based clone matching algorithm used to determine if an injected clone was detected by a subject tool. Next, we outline the individual steps of the evaluation phase that are executed for each pair of mutant system and subject tool. Including: (1) building the mutant system, (2) executing the clone detection tool for the mutant system, (3) measuring the tool's unit recall for the injected clone. We conclude with an overview of the evaluation phase's final statistical performance reporting about the subject tools.

Configuration

The evaluation phase has the following configuration options listed below. These options are used to configure the subsume-based clone matching algorithm during the evaluation phase. The meaning and consequences of these parameters are explained in the subsequent sections.

Subsume Tolerance A relaxing tolerance for the subsume-based clone matching algorithm, allowing the tool to miss a number of lines from the start and end of a reference clone while still being considered to have subsumed it. Specified as a ratio of the size (measure in lines) of the selected code fragment of the injected clone in the mutant system of interest. Must be equal to or less than the mutation containment in order to require the subject tools to detect the clone-type specific mutant aspects of

the reference clones.

Required Clone Similarity The minimum clone similarity of a detected clone to be accepted as a match of a reference clone. Can be disabled by setting to 0.

The evaluation phase is also configured with the subject clone detection tools it is evaluating. For each subject tool, the framework requires the following information:

Name The name of the tool.

Description A description of the tool.

Tool Runner The subject tool's tool runner executable.

When a subject tool is registered with the framework, its details are added to the experiment's database, which assigns the subject tool an unique identifier. The name and description of the tool are used by the framework to allow the user to easily identify a subject tool in the performance report. The tool runner is used by the framework to execute the subject tool automatically, and implements the input and output specification expected by the framework. The subject tools are specified by the framework user prior to the execution of the evaluation phase.

Clone Matching Algorithm

The framework uses a subsume-based clone matching algorithm to determine if a given clone pair, C, reported by the subject clone detector for a mutant subject system sufficiently matches the reference clone pair, R, injected into that mutant system, for R to be considered as detected by C. The framework considers C as a sufficient match of R if it meets three criteria: (1) C subsumes R given the configurable subsume tolerance; (2) C captures the mutations in R, and therefore handles the clone-type specific differences in R; and (3) C is not an obvious false positive that subsumes R by chance, as determined by a clone similarity threshold. We overview these three requirements in detail, followed by a mathematical definition of the full algorithm.

(1) Subsumes the reference clone To be a successful match, the reported clone (C) must subsume the injected clone (R). In other words, there must exist a pairing of C's and R's code fragments such that the code fragments of C subsume those of R. A subsume-based match is used because the injected clone may be surrounded by additional cloned code due to the selection of injection location. The subject tool may report this larger clone which subsumes the injected clone.

Clone detection tools may not report the reference clones perfectly. In particular, off-by-one line errors are common [13]. For this reason a subsume tolerance can be specified, allowing the tool to miss a number of lines at the start and end of the reference clone while still being considered to have subsumed it. This is controlled by the subsume tolerance configuration, a ratio. The number of lines that can be missed at the start and end of the reference clone's code fragments is equal to the subsume tolerance configuration (a

percentage) multiplied by the length (in lines) of the selected code fragment of the injected reference clone, rounded down to the nearest integer. For example, if the injected clone has a selected code fragment 10 lines long, and the framework user specified a subsume tolerance of 15%, then the tool is allowed to miss the first and last 1 lines ($\lfloor 10 * 0.15 \rfloor = \lfloor 1.5 \rfloor = 1$) of the clone. This method of specifying the subsume tolerance (in lines) is done to consider differences in clone size as well as to accommodate the second criteria.

(2) Handles the clone-type specific differences in the clone A goal of this framework is to evaluate how well the tools detect clones with particular types of differences, i.e., with different editing activities (mutations) and of different clone types. It is therefore required that the reported clone capture the mutations in the reference clone to be accepted as a successful detection of the reference clone. It is not sufficient for the tool to detect only the identical portions of the clone, it must also handle the introduced differences. For example, it is not sufficient for a subject tool to report only the identical regions of a Type-2 clone, it must handle and report the Type-2 differences as part of the overall clone.

This requirement is enforced using the subsume tolerance. During the generation of the reference corpus, the framework user specifies a mutation containment. This is the minimum distance of the mutation from the edges of the selected code fragment, specified as a ratio of the size of the selected (original) code fragment in lines. So long as the subsume tolerance is set equal to or less than the mutation containment, the clone matcher will only accept a reported clone that contains the mutation(s). The framework enforces this relationship between the two configuration parameters to ensure the subject tools must handle the mutations.

(3) Is not an obvious false positive A problem with the subsume clone matching algorithm is it will accept any reported clone that trivially subsumes the injected clone, even if the reported clone is an obvious false positive. In practice, this should not be common, but could be caused if the tool contains a bug causing it to report line boundaries incorrectly or large selections of mostly dissimilar code as clones. To account for this, we also require the reported clone to not be an obvious false positive to be accepted as a match of the reference clone. The clone is considered a false positive if its clone similarity, as measured in Section 3.1.1, falls below the required clone similarity threshold both by line and by token. This is user configurable, but should be set low as to only filter the obvious false positives. Clone similarity is measured after Type-1 (strict pretty-printing) and Type-2 (identifier and literal) normalizations.

Mathematical Definition We now summarize the above clone matching algorithm mathematically in Equation 3.4, where C is a clone reported by the tool, R is the injected reference clone in the mutant system, t is the subsume tolerance, and s is the required clone similarity. The $sim()$ function is implemented as described in Section 3.1.1. The subsume function is evaluated as in Equation 3.5, where $C.f_1$ and $C.f_2$ are the code fragments of the reported clone, $R.o$ is the original code fragment and $R.m$ the mutant code fragment of the reference clone, and $T(t) = \lfloor t * R.o.length \rfloor$. Equation 3.6 defines if a code fragment f_1 subsumes code fragment f_2 given a tolerance of $T(t)$ lines.

$$match(C, R) = subsumes(C, R, t) \wedge sim(C) \geq s \tag{3.4}$$

$$\begin{aligned} subsumes(C, R, t) = \Big(& C.f_1.subsume(R.o, T(t)) \wedge \\ & C.f_2.subsume(R.m, T(t)) \Big) \\ \vee \Big(& C.f_2.subsume(R.o, T(t)) \wedge \\ & C.f_1.subsume(R.m, T(t)) \Big) \end{aligned} \tag{3.5}$$

$$\begin{aligned} f_1.subsume(f_2, \tau) = & \\ & (f_1.file = f_2.file) \wedge \\ & (f_1.startLine \leq f_2.startline + \tau) \wedge \\ & (f_1.endline \geq f_2.endline - \tau) \end{aligned} \tag{3.6}$$

Step 1 - Construct Mutant System

The first step in the evaluation phase process, as executed per subject tool and mutant system pair, is to construct the mutant system. The mutant system's specification is retrieved from the database. This specification references the selected code fragment and its mutant code fragment that comprise the injected clone, along with their respective injection locations. The mutant system is constructed by duplicating the subject system, and copying the code fragments into the specified source files at the specified positions. The mutant system is then ready to be analyzed by the subject clone detection tools.

While each subject tool must be executed for the same mutant systems, the framework deletes and reconstructs the mutant systems for each tool. Many tools leave behind analysis files which could interfere with another tool's execution, while a bug in a subject tool could cause changes to the mutant system. To ensure the evaluation is fair, each tool is given a fresh version of each mutant system.

Step 2 - Clone Detection

In this next step, the framework executes the subject tools for the mutant system. The framework is able to execute a subject tool and collect its detection report automatically. To do this, a tool runner must be implemented for the subject tool. A tool runner is an executable that implements a simple communication protocol between the framework and the subject tool. The tool runner wraps the subject tool in an

input/output specification that the framework expects. It is the responsibility of the experimenter or tool developer to implement this tool runner.

For a given mutant system, the framework executes the tool runner and passes it the following input parameters: (1) the location of the mutant system, (2) the installation directory of the subject tool, (3) the properties of the reference corpus, including: the minimum and maximum clone size, the minimum clone similarity, and the mutation containment, and (4) the clone type of the injected clone, and the mutation operators used to synthesize it. The tool runner uses this information to configure and execute the subject tool for the mutant system. The framework expects the tool runner to output a clone detection report that lists the clone pairs the tool found in the mutant system in a simple comma separated format. It is up to the tool runner to covert the subject tool's output format to this standardized format. The framework retains a copy of this clone detection report for evaluation.

The tool runner is free to make use of any of the input data provided to it to configure the subject tool for the mutant system. By implementing multiple tool runners, it is possible to evaluate a tool's performance for different usage scenarios. For example, a tool runner could be implemented that ignores the injected clone information (clone type and mutation operators). It would configure the tool for generic clone detection, and the recall measurements would reflect the general usage of the tool. Another tool runner could consider the injected clone information, and configure itself for targeted detection of the injected clone. This result would be beneficial for tools which are highly configurable, especially towards the detection of specific clone types. The recall measurements would reflect highly targeted clone detection by the tool.

Step 3 - Measuring Unit Recall

Next, the subject tool's detection report is analyzed to evaluate its unit recall for the injected clone in the mutant system. A subject tool is given a unit recall of 1.0 for a particular mutant system if it successfully detects the injected clone, or 0.0 if it does not. To successfully detect the injected clone, the subject tool must report a clone that: (1) subsumes the injected reference clone within a given tolerance (2) captures the clone-type defining mutation in the reference clone, and (3) is itself not an obvious false positive that happens to include the reference clone. These conditions are evaluated by the clone-matching algorithm described in Section 3.2.3. The framework increments through the subject tool's clone detection report until a clone pair is found that is determined to be a successful match of the reference clone by the clone matching algorithm (unit recall = 1.0), or the end of the report is reached (unit recall = 0.0).

The measured recall depends on the configuration of the clone-matching algorithm, including the subsume tolerance and required clone similarity, as discussed earlier (Section 3.2.3). The subsume tolerance defines what ratio of the reference clone the tool can miss from the start and end of the clone while still being considered to have subsumed it. This cannot be set higher than the mutation containment configuration (of the generation phase) to ensure a detected clone is only considered a match of the reference clone if it captures the clone-type specific mutation operators. A higher subsume tolerance is more flexible, allowing

the tool to miss more of the clone while still being considered a match, favoring the tools in the evaluation. It acknowledges that when a developer uses the clone detection results, the detected clone is sufficient for them to identify the clone, as well as the missed lines. A lower threshold is more strict with the tools, expecting a more perfect capture. This is important for automated tasks that cannot recognize the missed portions of the clone. The subsume tolerance can also be completely disabled. The framework allows recall to be efficiently measured with multiple subsume tolerances. The framework user can then compare and interpret the differences in measured recall for different strictness in the capture.

The required clone similarity is used to judge if a detected clone is an obvious false positive, and should not be considered a match of the reference clone, even if it happens to subsume it. However, selecting a threshold for indicating obvious false positive clones can be rigid. To overcome this, the framework supports multiple efficient executions of the evaluation phase using different threshold configurations. For example, recall may be measured for a require clone similarity of: 0%, 50%, 60%, 70%.

Using a threshold of 0% disables this aspect of the clone-matching algorithm. A 50% threshold rejects a reported clone if it shares less than half of its syntax after normalizations. A higher threshold is more strict when measuring recall. The measurement can be compared against different thresholds, and the results interpreted. For example, if a tool's recall drops significantly between a 0% threshold (disabled) and 50% (weak threshold), this indicates the tool is capturing (subsuming) the reference clones, but its reporting of the reference clones contains a lot of additional dissimilar code, even after heavy normalization. If recall remains unchanged as the required clone similarity is increased from 0% to the minimum clone similarity of the generated clones (70% for example), this indicates that the tool is both capturing the reference clones and reporting clones that are highly similar syntactically. This threshold cannot be set higher than the minimum clone similarity threshold used in the generation phase. Note that the similarity threshold only affects the validation of Type-3 clones as Type-1 and Type-2 clones have 100% similarity after normalization.

Performance Reporting

The evaluation phase concludes by producing an evaluation report of the subject tools' recall performances for the generated reference corpus. For each tool, the framework reports its recall per mutation operator, per mutator, per clone type, as well as across the entire reference corpus. Summary values are calculated by averaging the unit performances across all mutant systems containing an injected clone part of that summary set. Per mutator performance is calculated by averaging the unit performance across all mutant systems containing an injected clone produced by that mutator. Per mutation operator performance averages the unit performance for all mutant systems containing an injected clone with at least one application of the mutation operator. While per clone type performance averages the unit performance for all mutant systems containing an injected clone of that clone type, as determined by the mutator used.

3.3 Using the Framework

The framework is operated by a menu-based command-line interface. The user uses this interface to create, load, configure and execute clone detection tool benchmarking experiments. A command-line interface was chosen as it allows the framework to be easily and efficiently executed and monitored remotely. Command-line applications can also be easily scripted and logged using input and output redirection. A menu-based user interface was chosen as it makes the framework very simple to use. The user is presented with a menu populated with the actions they can take at the current stage of their experiment. A help option is always available, which provides documentation on the available actions. An example of this interface is shown in Figure 3.6. This particular menu allows the user to configure the generation phase of their experiment.

```
--------------------------------------------------------------------------------
Stage 1/5: Generation Phase Setup
--------------------------------------------------------------------------------
Experiment: /home/user/MyExperiment/
--------------------------------------------------------------------------------
[1]: Configure Mutators and Mutation Operators.
| |
[2]: Review Generation Configurations.
[3]: Set Clone Granularity.
[4]: Set Clone Size.
[5]: Set Minimum Clone Similarity.
[6]: Set Mutation Containment.
[7]: Set Injection Number.
| |
[s]: Begin Generation Phase.
| |
[h]: Help.
[x]: Close Experiment.
--------------------------------------------------------------------------------
:::: _
```

Figure 3.6: Sample UI Menu

An experiment proceeds sequentially through five stages: (1) generation phase configuration, (2) generation phase execution, (3) evaluation phase configuration, (4) evaluation phase execution, and (5) results. In general, experiments progress through these stages sequentially. The exception is that an experiment in stage 5 may be returned to stage 3. This allows the user to reconfigure the evaluation phase, including: the addition, removal or modification of subject tools and reconfiguration of the clone matching algorithm and automatic clone validator. The experiment is then brought back to stage 5 via stage 4. Subsequent executions of the evaluation phase reuses clone detection reports and evaluated unit recall performances unaffected by the configuration changes.

Experiments are persisted to disk, and do not need to be completed within a single execution of the framework. The experiment may be closed during any of the interactive stages (1, 3 and 5). Closed experiments can then be relocated, or even exported to another computer, and be loaded later for continuation.

Experiments are self contained within their data directories, and can be duplicated. The user can fork their experiment by duplicating its directory. The user may wish to do this if they want to re-execute the evaluation phase for different settings or tools without modifying the original experiment. They simply duplicate the original, load the copy and proceed. The user might duplicate an experiment at the start of stage 3 (just after the generation phase) in order to archive the generated reference corpus for future use. When evaluating multiple subject tools for the same reference corpus, execution time can be decreased by duplicating the experiment after the generation phase and executing the evaluation phase for each tool independently on different computers.

Experiments may be shared with other users by sending them a copy of the experiment's data directory. By sharing an experiment, other users can view the results first hand, modify and re-execute the evaluation, or even extend the experiment with additional subject tools. The primary benefit of sharing experiments is that generated reference corpora can be shared amongst users. Ideally, a corpus is shared by providing a copy of the experiment at the start of stage 3 (i.e., after the generation phase but before subject tools have been added and the evaluation phase configured). However, the framework allows the removal of evaluation data from an experiment, so it is possible to transition any experiment in stage 3 or beyond to a state equivalent to the very start of stage 3.

We now overview the stages of the experiment, including the creation of a new experiment:

3.3.1 Experiment Creation

From the root menu, the user chooses to create a new experiment. The framework prompts the user for the programming language of the experiment (Java, C or C#), as well as the source code repository and subject system to be used. Once initialization is completed and confirmed by the user, the framework imports the repository and subject system into the experiment directory, and proceeds to the first stage of the experiment. Alternatively, the root menu provides the option to load an existing experiment, which resumes from the stage it was in when closed.

3.3.2 Stage 1 - Generation Phase Setup Stage

In this first stage, the user configures the generation phase. This includes the mutation operators, the mutators, and the generation parameters, including: clone granularity, clone size, minimum clone similarity, and injection number. The framework provides defaults and assistance in choosing these configurations. By default, the framework configures one single-operator mutator for each of the 15 included default mutation operators. These mutators can be removed, and the user can specify their own by providing the mutation operator sequences for their custom mutators. The user may also register their own custom mutation operators for use in the mutators. Once they are ready, the user initiates the execution of the generation phase, and the experiment proceeds to the next stage.

3.3.3 Stage 2 - Generation Phase Execution Stage

During this stage, the framework executes the generation phase, using the configuration specified in the previous stage, as outline in Section 3.2.2. This stage must be executed without interruption; it is not possible for the user to close or interact with the experiment during this stage. The framework outputs a detailed generation log so the user can monitor its progress.

3.3.4 Stage 3 - Evaluation Phase Setup Stage

In this stage, the user configures the evaluation phase, including: the participating subject tools and the clone matching algorithm. Stage 3 can be entered either from stage 2 (the generation phase) or stage 5 (results). In the former case, the experiment is being configured for its first execution of the evaluation phase, and no evaluation data exists. In the latter case, the subject tools are already fully evaluated. In this case, this stage is used to extend and/or re-configure an additional execution of the evaluation phase. The subject tools' clone detection reports and unit recall performances are re-used during subsequent executions of the evaluation phase, unless invalidated by a configuration change.

During this stage, the subject tools can be added, removed or reviewed. To add a subject tool, the user gives it an identifying name and description, and provides the framework the tool's installation location and its tool runner. When a subject tool is removed, its evaluation data (clone detection reports, evaluated unit performance) from any previous executions of the evaluation phase is deleted from the experiment. The user may also instruct the framework to delete a subject tool's evaluation data, including its previously collected clone detection reports and evaluated unit recall performances. The user may want to do this if they have modified the subject tool or its tool runner (e.g., the configuration of the subject tool for the experiment). Alternatively, the user can add another instance of the subject tool to the experiment with a different version of the tool runner. This way they can evaluate the tool for a different configuration without deleting the existing evaluation data.

The user must also configure the clone-matching algorithm, including the subsume tolerance and required clone similarity. The framework provides defaults and recommendations for these values. Changes to these values may necessitate the deletion of existing unit recall evaluation data for subject tools evaluated during previous executions of the evaluation phase. The unit recall data is deleted if the subsume tolerance or the minimum clone similarity is reconfigured The framework warns the user of these consequences before the changes are made. Alternatively, the user can duplicate the experiment and re-configure the copy so they don't lose their previous evaluation data.

3.3.5 Stage 4 - Evaluation Phase Execution Stage

During this stage, the framework executes the evaluation phase, per its configured in the previous stage, as outlined in Section 3.2.3. It does not re-execute a tool if its clone detection report already exists for a

mutant system from a previous execution of the evaluation phase. Likewise it does not re-evaluate unit recall if these measurements already exist. The previous stage deletes the existing results if configuration changes invalidate them.

The framework outputs a detailed evaluation log so the user can monitor the progress of the evaluation phase. The log is updated for each subject tools' evaluation of each mutant system. Unit recall is shown as it is evaluated. There is no option for closing the experiment during this phase. However, the framework can recover the experiment should it be terminated prematurely, either intentionally by the user or unexpectedly by a system error or power loss. When next opened, the experiment will resume from Stage 3, but retains any of the clone detection reports collected and unit performances evaluated during the previous execution. The user can then resume the evaluation phase.

Depending on the number of subject tools and their characteristics, the number of mutant systems, and the size of the mutant systems, the execution of the evaluation phase may take a significant amount of time. This time can be reduced by executing the evaluation phase for each subject tool in parallel. This can be done by closing the experiment in stage 3, duplicating it onto multiple computers, and executing stage 4 for each tool individually using the same evaluation phase configurations. Multiple experiments can be executed on the same computer when the subject tools do not require the full system resources. Once the evaluation phase is complete, the experiment proceeds to the results stage.

3.3.6 Stage 5 - Results Stage

Once the experiment reaches the result stage, the experiment is complete. Each of the subject tools have been evaluated for the generated reference corpus. The average recall performances of the subject tools can be viewed within the application. Average performance is summarized per clone type, per mutator, and per mutation operator. A full evaluation report, which also includes the configurations of the generation and evaluation phases, can be generated and saved to a file. From the results stage, the user can return their experiment to the evaluation phase setup stage. This allows the user to fully re-configure and re-execute the evaluation phase.

3.4 Limitations

The current version of the Mutation Framework supports the synthesis of function and block granularity clones in the Java, C and C# languages. However, the framework could be extended to support additional granularities and languages. New granularities require the implementation of code-fragment extraction, verification and injection code. This could be built by adapting the existing logic, which is implemented on our TXL-based distribution, and would not require significant re-implementation efforts. Supporting a new language requires providing a TXL-based grammar for that language, and implementing the granularity-specific code. The provided mutation operators can work for most procedural languages with C-like syntax

with minor modification, requiring just a simple TXL-based token grammar for that language. Procedural languages without C-like syntax might require re-implementation of the mutation operators. Other types of programming languages might require the design of new kinds of mutation operators specific to the domain. For example, the Mutation and Injection Framework has been adapted to Simulink models, which required mutation operators based on a taxonomy of the types of edits developers make on duplicated models [119]. Our goal in this work was to support the languages we find are most supported by the available clone detection tools.

A threat with the Mutation Framework is the synthetic clones may not be ones a real developer would create. The code fragments we randomly select for clone synthesis may not be ones a real developer would choose to clone. While the editing taxonomy guarantees our random mutations correspond to the types of edits real developers make on cloned code, it does not guarantee they apply edits a real developer would apply to the target code fragment. This is not a significant threat as the way clone detectors deal with these types of edits does not differ for edits performed by a real developer or synthetically. Some loss of realism is an accepted limitation of synthetic benchmarking, and we can overcome it by contrasting the Mutation Framework results against a real-world benchmark (such as our BigCloneBench).

We designed the framework to support as many clone detection tools as possible. The framework guarantees that the mutated and injected code is syntactically valid, but does not guarantee that the modified source files will compile. Therefore, clone detectors that rely on compiled code may not be compatible with the framework. This is not a limitation in the Mutation Framework concept or procedure, but its current implementation. The framework could be made compatible by adding a repair process which fixes compile errors after clone injection with additional code injection and modification. This would be very challenging, and was not considered during implementation of the framework as very few available clone detectors require compilable code.

During the generation process the framework can constrain the synthesized clones with a minimum similarity threshold measured by line and/or token after source normalizations. This constraint was included to help the user configure their subject clone detection tools appropriately for the generated corpus. A limitation here is that not all tools use line-based or token-based similarity metrics, and even those that do may measure similarity differently. Therefore, the user may still need to experiment with thresholds to find appropriately configurations for their subject tools. The limitation could be overcome by augmenting the framework with additional similarity metrics, including variations on line-based and token-based measurements. This is not a major limitation in the framework as clones can be reliably generated even with this constraint disabled.

3.5 Related Work

Some experiments have ignored recall, and simply measured precision by manually validating a small sample of a tool's candidate clones [38,52,70,76,82]. Others have tackled the recall problem by accepting the union of

multiple tools' candidate clones as the reference set, possibly with some manual validation [13,18,35,94,112]. For some experiments, very small subject systems were manually inspected for clones [18,71,110]. An ideal oracle could be made if all the pairs of code fragments in a subject system were inspected. However, this is not feasible except for toy systems. For example, when considering only clones between functions in the relatively small system Cook, there is nearly a million function pairs to manually inspect [137].

Bellon's Benchmark [13] is perhaps the most well-known clone benchmark. It is the product of Bellon et al.'s benchmarking experiment on tools contemporary to 2002 [13]. Their experiment measured the recall and precision of six clone detection tools for eight subject software systems. The reference corpus for their experiment was created by the manual validation of 2% (approximately 77 hours of manual effort) of the clones found by each of the participating tools. Clones which passed manual verification were added to the corpus, potentially with modifications to their line numbers as per the Bellon's judgment. Recall was reported as the ratio of the clones in the corpus that a tool was able to detect, and precision was reported as the ratio of the clones proposed by a tool that were accepted into the corpus after validation. The clones to be validated were chosen at random, and Bellon was kept unaware of which tool proposed a particular clone. The software used to run this experiment was released by the authors [13].

The primary difficulty with adopting Bellon's Benchmark as a unified benchmark is it is not convenient to use with additional detection tools beyond the original six. To add a tool to the framework, additional clone verification work must be performed on the results of the added tool and its contribution added to the corpus. If this is not done, the tool will have no representation in the reference corpus which may put that tool at a disadvantage for performance evaluation. We have demonstrated this in our previous work [128]. Even if a user of the framework does this verification work, they have to be very careful to validate using the same methodology as Bellon; a process that Baker found was not sufficiently documented to be repeated [9]. Additionally, the corpus is biased by the capabilities of the participating tools as it is limited to the clones the participating tools are capable of detecting. This also means the benchmark has no or poor representation of the types of clones the participating tools are unable or struggle to detect.

Roy et al. proposed the use of mutation analysis for clone detection tool benchmarking [107], including a proof of concept implementation and experiment to demonstrate its value [111]. The prototype framework was implement specifically for variants of a single clone detection tool (NiCad [110]), which allowed it to be rapidly implemented. The prototype took advantage of specifics of NiCad's operation and internal formats to remove significant complexities from the clone synthesis and tool evaluation procedure. The prototype was used in an experiment evaluating NiCad variants for a corpus of synthesized clones of types Type-1 through Type-4. The results of the experiment were consistent with known performance of the NiCad variants, demonstrating its success. However, significant challenges needed to be overcome to generalize the framework to all clone detection tools.

3.6 Conclusion

In this chapter, we presented the Mutation and Injection Framework: an automatic evaluation framework for measuring the recall and precision of clone detection tools. This framework uses an editing taxonomy for cloning to synthesize a reference corpus of artificial but realistic clones. The clone synthesis process mimics the copy, paste and modify cloning behavior performed by real developers. The framework enables a comprehensive reference corpus to be built without the need for manual candidate clone validation. The framework's capabilities extent to two clone granularities (function and block) and three popular programming languages (Java, C, C#). The framework user has many controls over the properties of the generated reference corpus. The framework automates the execution and evaluation of subject tools for the reference corpus. It provides a full statistical report on the performance of the participating subject tools. The framework is controlled by a simple user interface, that allows users to control and share their experiments and reference clone corpora.

Chapter 4

Evaluating Modern Clone Detection Tools

Many clone detection tools and techniques have been introduced in the literature, and these tools have been used to manage clones and study their effects on software maintenance and evolution. However, the performance of these modern tools is not well known, especially recall. In this chapter, we evaluate and compare the recall of eleven modern clone detection tools using our Mutation and Injection Framework. We compare these results against measurements by Bellon's Benchmark [13, 15], and its variants [94, 128], in order to comment on the state of the existing clone benchmarks. We compare the benchmark results against our knowledge and expectations of the subject clone detectors in order to comment on the accuracy of the benchmarks.

Bellon's Benchmark [13] was created for an experiment that compared the performance of six tools contemporary to 2002. It includes a corpus of curated clones mined from 2% of the output of the participating tools. Murakami et al. [94] extended Bellon's Benchmark to improve the correctness of its type 3 recall measurement by making the benchmark gap aware. They manually identified the gap lines in Bellon's type 3 clone references, and modified the benchmark's clone matching metrics to ignore these gap lines. As part of this work, we propose a modification to Bellon's *ok* clone matching metric that improves its accuracy and corrects a fault.

We evaluate the clone detectors using three versions of Bellon's Benchmark, including: (1) Bellon's original version, (2) Murakami et al.'s [94] gap aware extension of the benchmark, and (3) our modification of the *ok* clone matching metric. We also evaluate the tools' recall using our Mutation Framework. Of concern is the accuracy of Bellon's Benchmark, as its reference data is based on clones detected by tools over a decade old (2002). These clones may not be compatible with modern clone detection preferences such as scope, granularity, or what constitutes a true positive clone. We evaluate our confidence in both benchmarks by (1) checking for anomalies in their results, (2) checking for agreement between the benchmark, and (3) checking for agreement with our expectations. We also compare our results with Bellon's Benchmark against those of Bellon et al.'s [13] experiment. Our expectations of the tools' recall are flexible and researched, including contact with some of the tool developers. While expectations may contain inaccuracies, they define our confidence in a benchmark's results. By comparing our expectations with two benchmarks, we can get a good idea of the tools' capabilities.

We found that the Mutation Framework measures high recall for many of the tools. Particularly, it

suggests that ConQat, iClones, NiCad and SimCad are very good tools for detecting clones of all types. Many of the other tools also perform well. Clone detection users and researchers can consider these results, along with the features of the tools, to decide which tool is right for their use case. We find strong agreement between the Mutation Framework and our expectations, and suggest it is a good solution for measuring the recall of modern tools. Bellon's Benchmark frequently disagrees with our expectations and the Mutation Framework, often measuring considerably lower recall. We found anomalies in its results, including when we compare it against Bellon's original experiment. Our findings suggest that Bellon's Benchmark may not be accurate for modern tools, and that an updated corpus built by modern tools is warranted.

This chapter is based upon our manuscript [128] "Evaluating Modern Clone Detection Tools" published by myself and Chanchal K. Roy and in the Research Track of the International Conference on Software Maintenance and Evolution (2014), ©2014 IEEE. I was the lead author of this paper and study, under the supervision of my supervisor Chanchal K. Roy. The publication has been re-formatted for this thesis, with modifications to better fit this thesis.

This chapter is organized as follows. We provide a summary of Bellon's Benchmark and its variants in Section 4.1. We detail our experimental setup in Section 4.2, including the subject tools and their configurations. Then in Section 4.3 we analyze and discuss the results of the experiment. In Section 4.4 we discuss the threats to the validity of our results, and we conclude this work in Section 4.5.

4.1 Bellon's Benchmark

Bellon's Benchmark is a product of Bellon et al.'s [13] clone benchmarking experiment, which measured the recall of six contemporary (2002) tools for four C and four Java systems. The benchmark uses a reference corpus of real clones built by Bellon's manual verification ("oracling") of 2% of the 325,935 candidate clones detected by the tools. We use three variants of this benchmark including the original, Murakami et al.'s [94] gap-aware extension, and our version with an modified *ok* clone matching metric.

Bellon typified and added to the corpus only the true positives clones, as per his judgment, possibly with improvements to the clones' boundaries. This process was not formally specified, but from the experiment's publication [13], and from Baker's analysis of the experiment [9], we see that Bellon followed a number of rules: (1) minimum clone size of six lines including comments, (2) clone fragments may not start or end with comments, (3) clones must be replaceable by a function, (4) clones must be of the first three clone types, although Bellon additionally allowed type 2 clones to contain differences in expressions, (5) boundaries of accepted clones were expanded to the maximal size for their clone type, (6) clones capturing repetitive regions were left in the reporting style of the reporting tool. Due to disagreement over type 3 similarity requirements, no formal specification was used, and was instead left to Bellon's judgment.

Bellon's Benchmark automatically measures recall by mapping each of the clones detected by a tool (**candidates**) to one of the clones in the corpus (**references**). The mapping is produced using two clone

matching metrics, the *ok* and *good* values, which measure how well two clones match with a value between 0.0 (total mismatch) and 1.0 (exact match). The benchmark maps each candidate to the reference that maximizes its *ok* and *good* values, with *good* taking precedence as the stricter metric. A candidate is considered an *ok* match of the reference it is mapped to if its *ok* value exceeds some given threshold p, similarly for *good* match. The benchmark reports *ok* recall and *good* recall as the ratio of the references that the tool captures by the *ok* and *good* matches, respectively.

The *ok* metric is shown in Eq. 4.1, and measures how well clone candidate C matches reference R. F_1 and F_2 are a clone's first and second code fragments, ordered by file name, start line, and then end line. The *ok* metric is based on the contain metric, Eq. 4.2, which measures the ratio of F_A that is contained by F_B. For example, if both fragments are in the same file, and F_A includes lines 3 through 12 (inclusive) and F_B includes lines 7 through 14, then $contain = \frac{6}{10}$. The *ok* metric is the minimum containment of F_1 and F_2. It measures this for the optimal containment direction (C contains R, or R contains C) per fragment.

$$\begin{aligned} ok(C,R) = min(&max(contain(C.F_1, R.F_1),\\ &\quad contain(R.F_1, C.F_1)),\\ &max(contain(C.F_2, R.F_2),\\ &\quad contain(R.F_2, C.F_2))) \end{aligned} \tag{4.1}$$

$$contain(F_A, F_B) = \frac{|F_A \cap F_B|}{|F_A|} \tag{4.2}$$

The *good* metric is measured as in Eq. 4.3. It is based on the overlap metric, Eq. 4.4, which measures the ratio of the unique source lines in F_A and F_B that are in both fragments. For example, if both fragments are in the same file, F_A includes lines 1 through 10 (inclusive), and F_B includes lines 5 through 15 (inclusive), then $overlap = \frac{6}{15}$. The *good* metric is the minimum overlap of the candidate's and reference's first and second fragments.

$$\begin{aligned} good(C,R) = min(&overlap(C.F_1, R.F_1),\\ &overlap(C.F_2, R.F_2)) \end{aligned} \tag{4.3}$$

$$overlap(F_A, F_B) = \frac{|F_A \cap F_B|}{|F_A \cup F_B|} \tag{4.4}$$

Gap Aware Version. Murakami et al. [94] suggest that type 3 recall can be measured more correctly by ignoring the gap lines in the type 3 references when evaluating the *ok* and *good* metrics. A tool is then evaluated for how well it matches only the cloned lines in a type 3 reference. To enable this, they manually inspected Bellon's type 3 references and identified their gap lines. The *ok* and *good* metrics are then modified to discard the reference's gap lines. Specifically, $C.F_1$ is replaced with $C.F_1 - G_1$, $R.F_2$ by $R.F_2 - G_2$, and

similarly for $R.F_1$ and $C.F_2$ where G_1 and G_2 are the gap lines in the reference's first and second code fragments.

Our *Better-OK* Version. The *ok* match requires that either the candidate's or the reference's code fragments contain some minimum ratio of the other, using the containment direction per fragment that maximizes this ratio. The critical flaw in the *ok* metric is that it accepts either containment direction. For benchmarking, we should only be interested in if the candidate contains some minimum ratio of a reference. Candidates that are contained by a reference may be a very poor detection of that reference. For example, consider a 30 line (per fragment) reference clone, and a 6 line candidate whose fragments are fully contained by the reference. This candidate has an *ok* metric of 1.0 for the reference, or a perfect *ok* match. The same is true even if the reference is 100 lines. Obviously this is a very poor match of the reference, and should not be accepted. We modify Bellon's *ok* metric to only consider the ratio of the reference contained by the candidate, as shown in Eq. 4.5. We call this the *better ok* metric or *b-ok* for short. To evaluate *b-ok* recall, we replace Bellon's *ok* metric, but do not modify the *good* metric or the clone mapping procedure.

$$b\text{-}ok(C,R) = min(contain(R.F_1, C.F_1)), contain(R.F_2, C.F_2))) \tag{4.5}$$

4.2 Experiment

4.2.1 Bellon's Benchmark

We executed the tools for the benchmark's subject systems, and imported their results into the benchmark. We executed the benchmark's mapping and recall evaluation procedures using a clone matching threshold of 0.70. This is the value used in Bellon et al.'s [13] original experiment. The experiment was executed three times using Bellon's original clone matching metrics, Murakami's gap line metrics, and our 'b-ok' metric.

4.2.2 Mutation and Injection Framework

We evaluated the tools using two generated corpora, one Java and one C, of block granularity clones. For clone synthesis, we extracted code blocks from JDK6 and Apache Commons (Java), and the Linux Kernel (C). We injected the clones into IPScanner (Java) and Monit (C). For each corpus, we set the framework to randomly extract 250 code fragments, and mutate each using the 15 mutation operators, for a total of 3,750 clones. For each clone, 10 mutant systems were created using random injection locations, for a total of 37,500 unique mutant systems per corpus. We constrained the corpora to the following clone properties: (1) 15-200 lines in length per fragment, (2) 100-2000 tokens in length per fragment, (3) minimum 70% similarity measured by token and by line after type 1 and 2 normalization using a diff-based algorithm, and (4) mutations do not occur within the first and last 15% of a fragment (mutation containment). We selected the properties as the average default clone size and similarity defaults of the modern tools, which we believe

estimates modern clone preferences. Typically, clone detection tools are slower for smaller minimum clone sizes. We needed to use a larger clone size than Bellon's to make execution of the tools for 37,500 systems practical.

For the tool evaluation, we used a subsume tolerance of 15%, and a minimum clone similarity of 60%. By setting the subsume tolerance to the same value as the mutation containment, we guarantee that any candidate clone accepted as a match of a reference has captured all clone type specific differences (mutations) in the reference clone. For comparison against the Bellon's Benchmark results, we summarized recall per language and per clone type by averaging the per mutation operator results. Due to limited space, we do not report recall per mutation operator in this paper. We do not execute Deckard for our Java corpus as it does not support the needed language specification (Java 1.6).

4.2.3 The Participants

Eleven modern clone detection tools are investigated in this experiment. We used release date to judge the modernness of the tools. The oldest release of these tools was in 2006, while Bellon et al.'s [13] experiment was conducted in 2002. The participating tools are listed in Table 4.1. Ideally, we would have included the tools of Bellon's original experiment as participants of our Mutation Framework benchmark to compare the results against Bellon et al.'s original experiment. However, Bellon's publications [13] [15] do not list the versions of the six participants, nor their configurations. Most of these tools are no longer available, or the available versions are now significantly updated (i.e., modern tools).

Configuration. Our goal is to benchmark the performance of these tools from a user perspective. We want the results to represent what an experienced user would receive for their own systems. An experienced user has explored a tool's documentation and is comfortable modifying the default settings as required for their use case. To emulate this user, we configured the tools by considering: (1) the tool's default settings, (2) the tool's documentation, and (3) the properties of the benchmark, including minimum clone size and clone types. For settings that are not well documented, we experimented with the tool to find an appropriate value. We avoided over-configuring or over-optimizing the tools for the benchmark, as a user would not be able to do this for their own software systems. We also avoided configuring the tools in a way that would maximize recall at the sacrifice of precision. Generally, we configured the tools for a benchmarks' minimum clone size, and enabled type 2 normalization features. The tool configurations for each benchmark are summarized in Table 4.1. Different configurations are required due to differences in the benchmarks' minimum clone size. Since the properties of Bellon's corpus are not well known, we used more permissive settings with it.

Different configurations may result in better or worse recall for these tools. Wang et al. [139] refer to this as the confounding configuration choice problem. They propose the use of a genetic algorithm for finding tool configurations that optimize the tools' agreement on what is and isn't clone code in a software system. Using Bellon's Benchmark, they demonstrate that their configurations have a higher recall than the tools' default configurations. In general, compared to the default settings, their algorithm reduced minimum clone

Table 4.1: Participating Tools: Our Expectations and Configurations

Tool	Language	†Expected Recall (Type) 1	2	3	Configuration for Bellon's Corpus	Configuration for Mutation Framework
CCFinderX 10.2.6.4 [58]	Java C	◕	◕	○	min. size: 25 tokens, min. token types: 6	min. size: 50 tokens, min. token types: 12
ConQat 2012.9 [57]	Java	◕	◕	◕	min. size: 6 lines, max. editing distance: 3, max. gap ratio: 0.30	min. size: 15 lines, max. editing distance: 3, max. gap ratio: 0.30
CPD 5.0.4 [99]	Java	◕	◕	○	min. size: 30 tokens, literal/identifier normalization	min. size: 100 tokens, literal/identifier normalization
CPD 5.0.4 [99]	C	◕	○	○	min. size: 30 tokens	min. size: 100 tokens
CtCompare 3.2 [133]	Java C	◕	◕	○	min. size: 30 tokens, max. isomorphic relations: 6	min. size: 100 tokens, max. isomorphic relations: 3
Deckard 1.2.3 [53]	Java C	◕	◕	◕	min. size: 30 tokens, 5 token stride, min. 90% similarity	min. size: 100 tokens, 4 token stride, min. 85% similarity
Duplo 0.2 [32]	Java, C	◑	○	○	min. size: 6 lines, min. characters/line: 1	min. size: 15 lines, min. characters/line:1
iClones 0.1.2 [41]	Java C	◕	◕	◕	min. size: 30, min. block size: 10, all transformations	min. size: 100, min. block size: 20, all transformations
NiCad 3.4 [110]	Java C	◕	◕	◕	clone size: 4-2500 lines, blind renaming, literal abstraction, function and block clones, max. 30% dissimilarity	clone size: 10-2500 lines, blind renaming, literal abstraction, function/block clones, max. 30% dissimilarity
Scorpio 2011 [46]	Java	◕	◕	◕	min. size: 6 statements, normalize identifier/literal to type	min. size: 15 statements, normalize identifier/literal to type
SimCad 2.2 [136]	Java C	◕	◕	◕	consistent identifier renaming, function/block clones	consistent identifier renaming, block clones
Simian 2.3.34 [44]	Java C	◕	◕	○	min. size: 6 lines, normalize literals/identifiers	min. size: 15 lines, normalize identifiers/literals

†e.g., ◕◑◔ = 75% Type 1 Recall, 50% Type 2 Recall, 25% Type 3 Recall

size and enabled type 2 normalization features. Our strategy altered the default configurations in a similar way, although our settings are a little more cautious to prevent loss of tool precision. We also found that that our targeted configurations perform better than the tools' default settings with Bellon's Benchmark. Some of the tools' default configurations were optimized for demonstration use (short execution time). Our configurations may be more appropriate for benchmarking as configurations that optimize agreement between the tools may restrict the individual tools' unique detection characteristics and strengths.

Recall Expectations. Before we executed the benchmarks, we evaluated our expectations of each tool's recall, which are summarized in Table 4.1. We assigned expected recall in 25% increments, starting at 0% but capped at 90%. We consider a measured recall to agree with our expectation if it is within ±12.5% of the expected value. This strategy gives our expectations flexibility, as our expectations are educated estimates. If agreement is found, then we are confident that the expectation and benchmark are correct. Otherwise, we suspect that either our expectation and/or the benchmark is inaccurate.

We chose our expectations by consulting the tools' documentation, publication, and literature discussion [104, 108]. For type 1 and 2 recall, we considered the normalization features directly or indirectly supported by the tools. For type 3 recall, we considered the tools' similarity metrics and recommended sensitivity. We also considered our experiences with these tools in our other studies. Where possible, we reached out to the tool developers for their opinions of our expectations. We were optimistic about the quality of the tools, and of the benchmarks' ability to evaluate them. Despite these efforts, the expectations may still contain inaccuracies or be controversial between clone researchers. This is why we use a generous window (25%) around the expectation when determining agreement. These expectations give us the ability to uniformly evaluate our confidence in the benchmark results.

4.3 Results

We present and discuss the performance of the tools as measured by Bellon's Benchmark in Section 4.3.1. We comment on observed differences between the *ok* and *good* metrics, and discuss some anomalies in the results. In Section 4.3.2 we discuss how type 3 recall changes when measured by Murakami et al.'s [94] gap aware extension of the benchmark, and the performance of the tools using our *b-ok* metric in Section 4.3.3. We compare our results with the modern tools against Bellon et al.'s [13] original experiment in Section 4.3.4. We then compare our results with the variants of Bellon's Benchmark against our expectations for these tools in Section 4.3.5. In Section 4.3.6 we present and discuss the recall of the tools as measured by the Mutation Framework, and compare them against our expectations. We compare the results of the two benchmark in Section 4.3.7. The recall measurements of the two benchmark are summarized in Figure 4.1, and compared against our expectations in Table 4.2. We summarize agreement between the results and our expectations in Table 4.6, and agreement between the benchmarks in Table 4.7.

4.3.1 Bellon's Benchmark Results - Original Benchmark

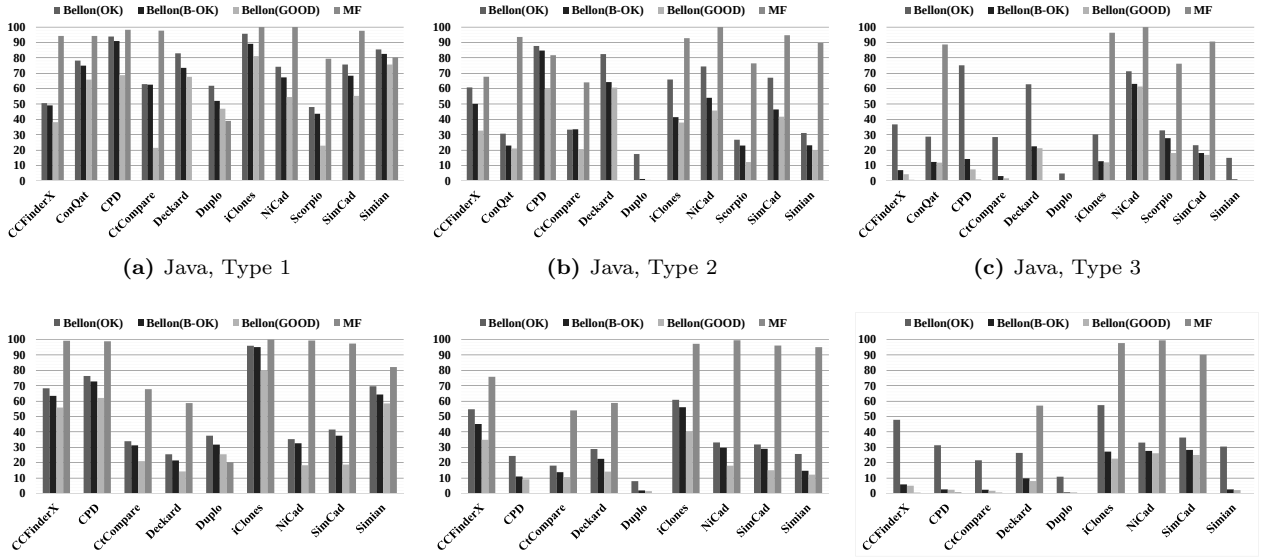
Java Type 1. Using the *ok* metric, most of the tools have a recall exceeding 70%, with CPD and iClones exceeding 90%. Scorpio performs poorly, detecting less than 50%, while CCFinderX only barely exceeds 50% recall. CtCompare and Duplo obtain fair results, with a little more than 60% recall. Many of these tools' recall drops considerably when the *good* metric is used. Only iClones and Simian exceed 70% recall with the good metric.

C Type 1. Most of the tools have poorer type 1 detection for C than Java. The exceptions are CCFinderX, which performs better for C, and iClones, which has comparable results for both languages. Only CPD, iClones and Simian exceed 70% recall with the *ok* metric. CCFinderX is just shy of 70% with the *ok* metric, while the remainder fall below 50%. Only iClones manages a recall over 70% with the *good*

Table 4.2: Expected Vs. Measured Recall: Mutation Framework (MF) and Bellon's Benchmark (*ok*, *b-ok*, *good* metrics)

	Tool	CCFX			ConQat			CPD			CtComp.			Deckard			Duplo			iClones			NiCad			Scorpio			SimCad			Simian		
	Clone Types	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
Java	Expected	◕	◕	○	◕	◕	◕	◕	◕	○	◕	◕	○	◕	◕	◕	◑	○	○	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	○
	MF	◕	◕	○	◕	◕	◕	◕	◕	○	◕	◕	○	—			◑	○	○	●	●	●	●	●	●	◕	◕	◕	◕	◕	◕	◕	◕	○
	ok	◑	◑	◑	◕	◔	◔	◕	◕	◕	◑	◔	◔	◕	◕	◑	◑	◔	◔	◕	◑	◔	◕	◕	◕	◑	◔	◔	◕	◑	◔	◕	◔	◔
	b-ok	◑	◑	◔	◕	◔	◔	◕	◕	◔	◑	◔	○	◕	◑	◔	◑	○	○	◕	◑	◔	◕	◑	◑	◑	◔	◔	◕	◑	◔	◕	◔	○
	good	◑	◔	◔	◕	◔	◔	◑	◑	◔	◔	◔	○	◕	◑	◔	◑	○	○	◕	◔	◔	◑	◑	◑	◔	◔	◔	◑	◑	◔	◕	◔	○
C	Expected	◕	◕	○	—			◕	○	○	◕	◕	○	◕	◕	◕	◑	○	○	◕	◕	◕	◕	◕	◕	—			◕	◕	◕	◕	◕	○
	MF	●	◕	○	—			●	○	○	◑	◑	○	◑	◑	◑	◔	○	○	●	●	●	●	●	●	—			◕	◕	◕	◕	◕	○
	ok	◑	◑	◑	—			◑	◔	◔	◔	◔	◔	◔	◔	◔	◑	◔	◔	◕	◑	◑	◔	◔	◔	—			◑	◔	◑	◑	◔	◔
	b-ok	◑	◑	◔	—			◑	◔	○	◔	◔	○	◔	◔	◔	◑	○	○	◕	◑	◔	◔	◔	◔	—			◑	◔	◑	◑	◔	◔
	good	◑	◑	◔	—			◑	◔	○	◔	◔	○	◔	◔	◔	◑	○	○	◕	◑	◔	◔	◔	◔	—			◔	◔	◑	◑	◔	◔

◕ = Indicates recall (0-100%) as ratio of pie filled (e.g., in this case, 75%).



(a) Java, Type 1 **(b)** Java, Type 2 **(c)** Java, Type 3

(d) C, Type 1 **(e)** C, Type 2 **(f)** C, Type 3

Figure 4.1: Measured Recall - Benchmark Results

metric, while CCFinderX, CPD and Simian have a recall greater than 50%.

Java Type 2. Only CPD, Deckard and NiCad exceed 70% recall by the *ok* metric. CCFinderX, iClones and SimCad exceed 50%, while the remainder fall below 35%. For the *good* metric, CPD and Deckard manage to maintain a recall around 60%, while the remainder fall below 50%.

C Type 2. Again, the tools generally perform worse for C clones. For the *ok* metric, only iClones and CCFinderX exceed 50% recall, while all the tools fall below 50% when the *good* metric is used.

Java Type 3. For the *ok* metric, only CPD and NiCad achieve a recall greater than 70%, with CPD just reaching 75%. Of the remaining tools, only Deckard exceeds 50%. With the *good* metric, NiCad achieves 60%, while the remainder fall below 25%. A notable anomaly is the type 2 detectors' sizable type 3 recall with the *ok* metric, especially CPD (75%). This is resolved by the good metric.

C Type 3. Only iClones exceeds 50% *ok* recall, with CCFinderX nearly meeting 50%. None of the tools have a *good* recall more than a little beyond 25%. Some of the type 2 detectors are achieving a type 3 *ok* recall, while their type 3 *good* recall is correctly near 0%. Type 3 detectors Deckard and NiCad have much better type 3 performance for the Java clones, while iClones and SimCad perform better for C.

In most cases, recall measured using the *good* metric is considerably lower than recall measured by the *ok* metric. Bellon's Benchmark suggests that the modern tools are bad at detecting a clone's precise boundaries. However, for the type 1 and 2 clones, this may be due to Bellon's oracling process. When he accepted a clone he would change the clone's boundaries (if needed) to the maximal size of its reported clone type [9]. It is possible that the accepted clone could have been contained within a larger clone of a higher clone type. Likely, the tools prefer to report the larger clone of the higher type. The *ok* metric (contain) will accept this larger clone as a match, but the *good* metric (precise capture) is likely to reject it. For this reason, the *good*

metric may be too strict for the type 1 and type 2 clones. However, our proposal of the *b-ok* metric shows why the *ok* metric is too permissive.

However, the *ok* metric may be too weak for measuring type 3 recall. A number of type 2 detection tools were receiving sizable type 3 recalls with the *ok* metric. Since these tools cannot detect type 3 clones, the *ok* metric must be permissive enough to sometimes accept clones which only capture the type 1 or 2 portions of the type 3 clone. When measuring type-specific recall, this is undesirable. The *good* metric was appropriately measuring near-zero recall for these tools.

Overall, we found that many of the tools performed well for type 1 Java clones, and a few had good performance for type 2 Java clones, when the *ok* metric is used. Performance was generally weaker for C clones, and most tools performed poorly for both language's type 3 clones. The type 2 clone detectors were surprisingly able to achieve a type 3 recall when the *ok* metric was used, while the type 3 detectors are struggling to detect type 3 clones, even when measured by the permissive *ok* metric.

4.3.2 Bellon's Benchmark Results - Murakami Extension

Murakami et al. [94] suggest that type 3 recall is more correctly measured when Bellon's *ok* and *good* metrics ignore the gap lines in the type 3 references. We found that ignoring the gap lines has minimal impact on the tools' *ok* and *good* type 3 recall. Compared to Bellon's original metrics, the tools' gap-ignoring type 3 recall has an absolute change of no more than $\pm 1.5\%$, with two exceptions. CPD's type 3 *ok* recall for Java has an absolute increase of 7.2%, and iClones's type 3 *good* recall for C an increase of 3.7%. Ignoring these outliers, the average absolute change was $\pm 0.48\%$. This means that it is extremely rare for these tools to fail to capture a type 3 reference due to reporting only the cloned regions but not the non-cloned regions.

Murakami et al. investigated this difference for NiCad, Scorpio and CDSW. Their experiment found significant differences in type 3 recall for Scorpio and CDSW. Their experiment agrees with ours that ignoring gap lines has negligible effect on NiCad's recall. Our disagreement over Scorpio may be due to how we handled Scorpio's output. Of our subject tools, Scorpio is unique in that it does not report code fragments as source line regions. It is PDG-based, and reports code fragments as sets of (possibly non-sequential) program elements. We converted these to continuous line regions using the source lines of the earliest and latest program elements as the start and end lines. Murakami et al. do not mention how they handled this in their experiment.

While ignoring gap lines had minimal effects in our experiment, Murkami's gap line data is still valuable. It can be used to evaluate the correctness of clone detection tools that identify gap lines in their reported clones. Bellon's Benchmark is designed to handle clone detection tools that report code fragments as continuous source line regions. Knowing the locations of gaps in the type 3 references may make it possible to adapt Bellon's Benchmark to support tools that report code fragments as discontinuous source line regions without (approximate) conversions of their output.

4.3.3 Bellon's Benchmark Results - The Better OK Metric

The recall of the tools using our *b-ok* metric are compared against Bellon's *ok* recall in Figure 4.1. The relative change in recall going from Bellon's *ok* recall to our *b-ok* recall is summarized in Table 4.3, with the tools grouped by the maximum clone type they are able to detect. We also average the relative change across the tools, including specifically for tools that support or do not support particular clone types. The tools' recall decreases when our *b-ok* metric is used, with the exception of a marginal increase in CtCompare's Java type 2 recall. The decrease in recall means that there are candidates reported by the tool that are 70% contained by references in the benchmark, but none of these candidates contain 70% of these references. Likely, the tool reported only a small (<70%) portion of these references, which is why our *b-ok* metric rejects them as a match. While recall generally decreased, an increase is possible (CtCompare) because Bellon's Benchmark maps each candidate to the reference that maximizes its *good* and then *ok* values. By replacing the *ok* metric by the *b-ok* metric, the mapping can change in such a way that more references are matched given the matching threshold (70%).

Tools lacking type 3 support lose the majority of their recall (81-100%), and similarly for tools lacking type 2 support (55-93%). This improves the anomaly we found in Bellon's *ok* recall for which the tools' have sizable recalls for clone types they don't support. With Bellon's *ok* metric, a tool could match a type 3 reference by reporting a candidate that captures even a minuscule type 1 or 2 region within the reference, or a minuscule type 1 region in a type 2 reference. Our improved metric will only accept these cases when the detected lower clone type region is at least 70% of the reference clone. This appears to be rare, as the average *b-ok* recall for clone types a tool does not support is 4% with a maximum of 15%.

On average, the relative decrease in recall is larger for higher clone types, considering only the tools that support the respective types. Type 3 recall reduction was significant for some of the type 3 tools, with ConQat, Deckard and iClones losing over half of their recall when our *b-ok* metric is used. Even these advanced tools are only reporting small (<70%) regions of the references that were matched by Bellon's *ok* match. What is unknown is if this is due to a deficiency in these tools, or disagreement with Bellon's definition of a type 3 clone, particularly the amount of dissimilarity allowed. The other type 3 tools had less significant reductions (12-16% relative decreases). Overall, our correction to Bellon's *ok* metric has a considerable effect on the measured recall.

4.3.4 Bellon's Benchmark - Modern Vs. Original Experiment

Our expectation is that clone detection has significantly evolved since Bellon's original experiment in 2002, especially for type 3 detection. In this section, we examine if Bellon's Benchmark supports this expectation by comparing our results with the modern tools against the results of Bellon's original experiment. In order to compare the two experiments, we calculated the average and maximum *ok* and *good* recalls across the tools of the individual experiments (Table 4.4).

Table 4.3: OK to BetterOK - Relative Change in Recall

		Java Clone Types (%)			C Clone Types (%)		
Tool		**1**	**2**	**3**	**1**	**2**	**3**
1	Duplo	-16	-93	-100	-15	-75	-94
	CPD	-	-	-	-5	-55	-92
2	CCFinderX	-3	-18	-81	-7	-18	-88
	CPD	-3	-3	-81	-	-	-
	CtCompare	-1	1	-89	-8	-24	-89
	Simian	-3	-26	-94	-8	-43	-91
3	ConQat	-4	-25	-57	-	-	-
	Deckard	-11	-22	-64	-16	-22	-63
	iClones	-7	-37	-58	-1	-8	-53
	NiCad	-9	-27	-12	-8	-10	-16
	Scorpio	-9	-14	-15	-	-	-
	SimCad	-10	-31	-22	-10	-9	-22
Avg: Tool Doesn't Support		-	-93	-89	-	-65	-91
Avg: Tool Does Support		-7	-21	-38	-9	-19	-39
Avg: All		-7	-27	-61	-9	-29	-68

Table 4.4: New Vs. Old Experiment

		OK Metric			GOOD Metric		
Java		**1**	**2**	**3**	**1**	**2**	**3**
AVG	**new**	73.6	52.5	37.2	54.4	32.1	14.1
	old	61.6	48.9	26.5	43.5	33.0	4.1
MAX	**new**	95.6	87.7	75.1	81.1	60.7	61.4
	old	94.9	85.3	61.6	67.3	48.9	7.5
C		**1**	**2**	**3**	**1**	**2**	**3**
AVG	**new**	53.8	31.7	32.8	39.3	17.3	10.4
	old	52.8	36.6	32.9	41.7	25.8	8.8
MAX	**new**	96.0	60.9	57.4	79.9	39.9	26.1
	old	85.7	79.8	68.3	79.0	68.1	22.2

Java. In general, the new tools outperform the old for the Java clones with both the *ok* and *good* metric. However, the average recall of the new tools is not as significant of an improvement as we expected. At most, the modern tools lead the old tools by 12%, and fall behind by 1% in one case. Considering the best performing of the new and old tools (maximum), the new tools perform consistently better. The increase in maximum recall is marginal with the *ok* metric, except for type 3 clones (+15%). The new tools have a considerably higher maximum *good* type 3 recall (+54%). This advantage is from NiCad, while the other modern tools had negligible *good* type 3 recall.

C. For the C clones, both the new and old tools have a very similar average recall by both metrics, with the old tools having up to 10% better recall for type 2 clones. Considering the best performing of the tools, the old tools mostly perform better, with the new tools having only a slight advantage in type 3 detection with the *good* metric (+3.9%).

The difference in type 3 recall between the old and new tools is strange. Only for the Java clones by the *good* metric does the best of the new tools outclass the old tools for type 3 detection. In the other cases,

the difference is only 3-15%, with the best of the newer tools performing worse for type 3 C clones by the *ok* metric. Type 3 clone detection has been an area of focus in clone detection since Bellon's original experiment, so we expected the new tools to perform much better than the old. This suggests that Bellon's corpus does not have sufficient type 3 representation to accurately judge these modern tools.

These two experiments are not exactly equivalent. The old tools have the advantage that Bellon's clone references are based off the clones the old tools detected. However, the modern tools have the advantage of up to a decade of clone detection research. Even if the the new tools did not contribute to the benchmark, they are the state of the art and should not have a problem detecting clones found by their predecessors.

It is interesting to compare CCFinderX to its direct predecessor, CCFinder, that participated in the original experiment (Table 4.5). We can reasonably assume that CCFinderX (2009) should be an improvement over CCFinder (2002). However, CCFinderX's recall is considerably lower than CCFinder's for all clone types and both metrics, with the exception of a 6% lead in Java type 3 *ok* recall. The exception is likely an anomaly, as both versions of CCFinder lack type 3 support. In the original experiment, CCFinder was executed for its default settings, while we executed CCFinderX with more permissive settings than its modern default. It is possible that CCFinderX's core algorithm is less aggressive in detecting clones, possibly to increase precision. Or perhaps CCFinderX's preferences of what constitutes a true positive clone has changed, and disagrees with Bellon's definitions. Having a previous version that contributed to the corpus, we expected CCFinderX to be somewhat attuned to the benchmark. That CCFinderX performs considerably worse than CCFinder suggests that clone preferences have changed, and that the modern tools cannot be accurately judged by Bellon's corpus.

4.3.5 Bellon's Benchmark Variants Vs. Expectations

In this section we compare the tools' recall as measured by Bellon's Benchmark, using both its original and our improved metrics, against our expectations for these tools. Since we created our expectations in 25% increments, we consider measured recall to agree with our expectations if their absolute difference is 12.5% or less. Measured recall is compared against our expectations in Table 4.2, and their agreement is summarized in Table 4.6.

Type 1. The *ok* recall agrees with our expectations for 6 of the 11 Java tools, but for only 2 of the 9 C tools. Three of these tools lose agreement when our *b-ok* metric is used. In the cases of disagreement, the recall measurements are generally considerably lower than our expectations. Only iClones (Java and C) and Duplo (Java) agree with our expectations with the *good* metric. The *good* recall of the remainder is considerably lower than our expectations. We expected these tools (with the exception of Duplo) to have a type 1 recall around 90%. These tools remove type 1 differences when they parse or preprocess input code. By configuring the tools with the benchmark's minimum clone size, it should be trivial for these tools to detect the simple type 1 clones. At the very least, the tools should detect the type 1 references as components of larger type 2 or type 3 clones, which would be accepted by the *ok* and *b-ok* metrics if not the *good* metric.

Table 4.5: CCFinder vs. CCFinderX

	OK			GOOD		
Clone Types	1	2	3	1	2	3
CCX-Java	50.5	60.7	36.7	38.2	32.7	4.3
CC-Java	88.0	85.3	30.9	42.5	48.9	6.3
CCX-C	68.3	54.7	47.8	55.8	34.9	5.0
CC-C	85.7	79.8	68.3	79.0	68.1	13.0

Table 4.6: Agreement Between Measured and Expected Recall, Mutation Framework (MF) and Bellon's Benchmark

	Tool	**CCFX**			**ConQat**		**CPD**			**CtComp.**			**Deckard**			**Duplo**			**iClones**			**NiCad**			**Scorpio**			**SimCad**			**Simian**			**% Agree**
	Clone Types	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	
Java	Expected	◕	◕	○	◕	◕	◕	◕	◕	○	◕	◕	○	◕	◕	◕	◑	○	○	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	○	-
	MF	⊗	○	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	—			⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	○	○	⊗	⊗	⊗	⊗	⊗	⊗	90.0%
	ok	○	○	○	⊗	○	○	⊗	⊗	○	○	○	○	⊗	⊗	⊗	⊗	○	⊗	⊗	○	○	○	○	○	○	○	○	○	○	○	⊗	○	○	30.3%
	b-ok	○	○	⊗	○	○	○	⊗	⊗	○	○	○	⊗	○	○	○	⊗	⊗	⊗	⊗	○	○	○	○	○	○	○	○	○	○	○	⊗	○	⊗	30.3%
	good	○	○	⊗	○	○	○	○	○	⊗	○	○	⊗	○	○	○	⊗	⊗	⊗	⊗	○	○	○	○	○	○	○	○	○	○	○	○	○	⊗	24.2%
C	Expected	◕	◕	○	—			◕	○	○	◕	◕	○	◕	◕	◕	◑	○	○	◕	◕	◕	◕	◕	◕	—			◕	◕	◕	◕	◕	○	-
	MF	⊗	○	⊗	—			⊗	⊗	⊗	○	○	⊗	○	○	○	○	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	—			⊗	⊗	⊗	⊗	⊗	⊗	74.1%
	ok	○	○	○	—			○	○	○	○	○	○	○	○	○	⊗	⊗	⊗	⊗	○	○	○	○	○	—			○	○	○	○	○	○	14.8%
	b-ok	○	○	⊗	—			○	⊗	⊗	○	○	⊗	○	○	○	○	⊗	⊗	⊗	○	○	○	○	○	—			○	○	○	○	○	⊗	29.6%
	good	○	○	⊗	—			○	⊗	⊗	○	○	⊗	○	○	○	○	⊗	⊗	⊗	○	○	○	○	○	—			○	○	○	○	○	⊗	29.6%

⊗ = Agree (±12.5%) ○ = Disagree ○◔◑◕● = Recall (0, 25, 50, 75, 90%)

It is strange that recall is further from our expectations in the C cases, despite the tools advertising full C language support.

Type 2. Recall measurements agree with our expectation for tools that do not support type 2 clones (Duplo, CPD for C), at least by the *b-ok* and *good* metrics. Of the tools supporting type 2 detection, CPD's (Java) agrees with our expectations by both the *ok* and *b-ok* metrics, as well as Deckard (Java) by only the *ok* metric. Otherwise, none of the tools' type 2 recalls agree with our expectations. Generally, their type 2 recall is considerably lower than our expectations. We expected many of these tools to have near 90% recall for type 2 clones. Most support the required type 2 normalizations (literal values, identifier names), which reduces type 2 detection to simple type 1 detection. It is therefore suspicious that not only do these results not agree with our expectations, but that the results do not mirror the type 1 results. This is at least partially due to Bellon's oracling process, which allowed type 2 clones to have an identifier or literal in one fragment be replaced by an expression in the other, as found by Baker [9]. The modern tools would consider these replacements to be near-miss gaps, and consider them to be type 3 clones. In this case the type 2 detectors would fail to report them, while the type 3 detectors may find them similar enough to report. However, this oracling error may not be too pronounced as the best performing Java type 2 detector is CPD, which lacks type 3 support.

Type 3. Considering the type 2 detectors, the *b-ok* and *good* recalls agree with our expectations (0% recall), with the exception of CPD for Java clones and the *b-ok* recall. The strong agreement is because these metrics generally will not accept a type 2 candidate as a match of a type 3 reference. With the

exception of Duplo, the type 2 detectors' type 3 *ok* recall does not agree with the expectation. As mentioned previously, the *ok* metric is allowing candidates detecting even only small regions of the type 3 clone as matches. Considering the type 3 detectors, none of the tools' type 3 recalls agree with our expectations. The only exception is Deckard whose Java type 3 *ok* recall agrees with our expectations. However, its *b-ok* and *good* Java type 3 recalls are considerably lower than our expectations. Considering Bellon's type 3 references were found by tools contemporary to 2002, and modern tools are considered to excel at near-miss clone detection, it is strange that they perform so far under our expectations. Perhaps the corpus simply does not have a large or diverse enough representation of type 3 clones to evaluate modern tools. Or perhaps Bellon's type 3 references disagree with the modern tool's type 3 clone preferences (e.g., scope, minimum similarity, true vs. false positive, etc.).

Recall measured by Bellon's Benchmark is generally lower than our expectations. A common belief in the clone community is that our clone detection techniques are very mature and have high recall. The disagreement between our expectations has two possible and potentially overlapping conclusions: (1) the modern clone detection tools are not as proficient as we believe, i.e., our expectations are incorrect, or (2) Bellon's Benchmark does not accurately measure the performance of modern tools. To gain further insight into this question, we consider the results of the Mutation Framework below.

4.3.6 Mutation Framework Results vs. Expectations

In this section we discuss the recall of the tools as measured by the Mutation Framework, and compare these against our expectations for the tools. As the granularity of our expectations was 25%, we consider the measured and expected recalls to agree if they have an absolute difference no greater than 12.5%. The tools' recall by the Mutation Framework are shown in Figure 4.1, and compared against our expectations in Table 4.2. Agreement with expectations is summarized in Table 4.6.

Type 1. The Mutation Framework measures a very high recall ($> 90\%$) for most of these tools across both languages. Scorpio's and Simian's recall is a little lower at 80%. These results agree with our expectations of the tools. Duplo has poor type 1 recall, as it doesn't normalize for formatting differences, which is within our expectations for Java but not for C. While CtCompare has strong recall for the Java clones, its type 1 performance was considerably weaker for the C clones, and outside of our expectations. Deckard's recall for C is also below our expectations, at 59%.

Type 2. The framework also measures very high recall (>90%) for most of the tools that support type 2 detection. CPD falls a little behind the top performers, with a Java recall of 81%. These results match our expectations. The framework correctly identifies that Duplo and CPD (for C) do not support type 2 clones, with a near 0% recall. CtCompare does not support literal value normalization and recommends limits on identifier normalization, so we are not surprised by its lower recall for Java (64%), although its recall for C (53%) is lower than anticipated. Scorpio's recall falls just outside our expectations, with a recall of 76%. CCFinderX is also less than our expectations, with 67% for Java and 76% for C. Deckard's type 2 recall

matches its type 1, 59% and is considerably less than we expected.

Type 3. iClones and NiCad have near-perfect recall (>95%) for both languages, while ConQat (89%) and SimCad (90%) also achieve very high recall. These results match our expectations for these tools. The framework correctly identifies the tools that lack type 3 support, with near 0% recall. The framework measures recall outside of our expectations for Scorpio (76%) and Deckard (56%).

Compared to the other type 3 detectors, Scorpio and Deckard have lower recalls. Notable is how consistent their recall is across the clone types. To prevent bias between recall measurements, the Mutation Framework uses the same original code fragments with each of the 15 mutation operators, and injects each of the 15 resulting clones at the same locations in the subject system. Therefore Scropio and Deckard may not have any deficiency for any particular clone type, but rather failed to detect these clones due to general deficiencies in its parser or detection algorithms.

Overall, there is strong agreement between our expected recall and the Mutation Framework's results. Agreement is found in 30 out of 33 Java cases, and 20 out of 27 C cases. In the cases of disagreement, the Mutation Framework consistently measured a lower recall. Strong agreement suggests confidence in the accuracy of the Mutation Framework. We did not notice any anomalies in the Mutation Framework's results.

4.3.7 Bellon's Benchmark vs Mutation Framework

In this section we directly compare the recall measurements of Bellon's Benchmark and the Mutation Framework. Since the two benchmarks were constructed differently (mined versus synthetic clones), we consider them to agree if the measured recalls are within 15%. Agreement between the benchmarks is show in Table 4.7. Despite the differences in their approaches, it is reasonable to expect the benchmarks to agree. The Mutation Framework tests the tools against a comprehensive (and empirically validated) taxonomy of the types of differences that can occur between clones. The base code fragments used for synthesis and injection locations in the subject system are randomly varied to ensure variety. We expect that if tools perform well for the Mutation Framework's synthesized clones, that this performance should transfer to clones naturally produced by developers.

However, in very few cases do these benchmarks agree. They agree on the type 1 recall of CPD (Java), Duplo, iClones and Simian, as well as the type 2 recalls of CCFinderX (Java), CPD and Duplo. For CCFind-

Table 4.7: Mutation Framework (MF) vs. Bellon's Benchmark (*ok*, *b-ok*, *good*)

	Tool	CCFX			ConQat			CPD			CtComp.			Deckard			Duplo			iClones			NiCad			Scorpio			SimCad			Simian			% Agree
	Clone Types	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	
Java	MF vs *ok*	○	⊗	○	○	○	○	⊗	⊗	○	○	○	○	-	-	-	○	○	⊗	⊗	○	○	○	○	○	○	○	○	○	○	⊗	○	⊗	23.3%
	MF vs *b-ok*	○	○	⊗	○	○	○	⊗	⊗	⊗	○	○	⊗	-	-	-	⊗	⊗	⊗	⊗	○	○	○	○	○	○	○	○	○	○	⊗	⊗	36.7%
	MF vs *good*	○	○	⊗	○	○	○	○	○	⊗	○	○	⊗	-	-	-	⊗	⊗	⊗	○	○	○	○	○	○	○	○	○	○	○	⊗	○	⊗	26.7%
C	MF vs *ok*	○	○	○	-	-	-	○	○	○	○	○	○	○	○	○	○	⊗	⊗	⊗	○	○	○	○	○	-	-	-	○	○	○	⊗	○	○	14.8%
	MF vs *b-ok*	○	○	⊗	-	-	-	○	⊗	⊗	○	○	⊗	○	○	○	⊗	⊗	⊗	⊗	○	○	○	○	○	-	-	-	○	○	○	○	○	⊗	33.3%
	MF vs *good*	○	○	⊗	-	-	-	○	⊗	⊗	○	○	⊗	○	○	○	⊗	⊗	⊗	○	○	○	○	○	○	-	-	-	○	○	○	○	○	⊗	29.6%

⊗ = Agree (±15%) ○ = Disagree

erX (type 2, Java) and Simian (type 1, C), this agreement is only with Bellon's *ok* recall. We have shown that the *ok* recall can be unreliable. The benchmarks agree in the cases where a tool does not support a particular clone type if either the *b-ok* or *good* recalls are considered. The benchmarks disagree in some of these cases when the *ok* recall is used, which supports our findings that the *ok* metric can lead to incorrect recall measurements for clone types a tool doesn't support. In all other cases, the frameworks disagree on the tools' recall. Generally, the Mutation Framework measures a higher recall in these cases. With Bellon's Benchmark, the tools generally had lower recall for C clones, but this is not common in the Mutation Framework results.

Disagreement between the benchmarks over NiCad and SimCad is not suspicious. These tools detect clones at the code block granularity: code that starts and ends with matched brackets, i.e., '{...}'. The Mutation Framework generates clones at this granularity to support more tools. Tools that search at a lower granularity (i.e., within code blocks) do not have a disadvantage with the Mutation Framework. However, NiCad and SimCad may fail to detect clones in Bellon's corpus that are much smaller than a code block. Despite this, NiCad has the top Java type 3 recall by Bellon's Benchmark.

It is particularly strange that the benchmarks disagree for type 1 and type 2 recall. When the Mutation Framework measures a high recall for these types, it has certified that the tool can handle all 10 of the variations in type 1 and type 2 clones from the clone taxonomy. Many of the modern tools received this certification. As per mutation analysis, the Mutation Framework mutates only a single random difference into the reference clones. The tools detect type 1 and type 2 clones by removing or normalizing these differences during parsing or preprocessing steps. Therefore, the tools should have no problems detecting type 1 and type 2 clones, no matter the density of the type 1 and type 2 features. It is odd that Bellon's Benchmark measures considerably lower type 1 and 2 recall for some of the tools that have very high recalls by the Mutation Framework. It is possible that this is due to changes in clone detection preferences between the 2002 tools and the modern tools. Detection preferences may include clone granularity, scope, what constitutes a true positive, what clones are useful to report, and so on.

The benchmarks do not agree on the type 3 recall of any of the type 3 tools, with the Mutation Framework consistently measuring a higher recall. The Mutation Framework shows that most of the type 3 detectors are able to handle the types of differences that can occur between type 3 clones. We constrained the Mutation Framework to generate clones with similarity no less than 70%. Bellon provided no specification for his type 3 clones. It may be that Bellon's type 3 clones contain a higher degree of dissimilarity, more than the tools allow, which would result in a lower recall from Bellon's Benchmark. It is also possible that Bellon's corpus does not have sufficient type 3 representation to accurately measure recall. Bellon's corpus was built using tools contemporary to 2002, when type 3 detection was not as well developed. However, is is strange that the modern tools are not able to detect more of the type 3 clones found by their "outdated" predecessors. This suggests that type 3 preferences have changed, and the modern tools target a newer specification.

The Mutation Framework has a much stronger agreement with our expectations than Bellon's Benchmark,

as shown in Table 4.6. The Mutation Framework agrees with our expectations in 90% (Java) and 74.1% (C) of the cases, while Bellon's Benchmark only agrees in 24.2-30.3% (Java) and 14.8-29.6% (C), depending on the metric used. We suspect that in the cases where neither tool agrees with our expectation, that our expectation is incorrect. Bellon's strong disagreement with our expectation, and the suspicions we raise about its results, suggest that it is not accurate for modern tools. The Mutation Framework's strong agreement with out expectations suggest that it may be a good solution for evaluating the modern tools. In cases where the Mutation Framework disagrees with our expectations, we suspect that our expectation is incorrect, and the Mutation Framework accurate.

4.4 Threats to Validity

There are three primary threats to the validity of this study. (1) Our expectations of the tools' recall may not be accurate. We maximized our accuracy by consulting the tools' documentation, publication, literature surveys, and developers (when available). Furthermore, we allowed a $\pm 12.5\%$ range around our expectation to compensate for some inaccuracy. We used these expectation ranges as a baseline for our confidence in the benchmarks. (2) The tool configurations may not be optimal. We created targeted configurations by consulting the tools' defaults and documentation, which is how the average user would configure the tools for their use cases. While other configurations might give higher recall, our configurations measure the recall the average user can expect. (3) The Mutation Framework uses artificial clones. However, these clones are generated using mutation analysis, which is a well established technique in other fields including software testing. The clones are generated using a comprehensive clone taxonomy empirically validated against real clones [108], so the generated clones should be realistic.

4.5 Conclusion

In this chapter, we compared the recall performance of eleven modern clone detection tools using our Mutation and Injection Framework and three variants of Bellon's Benchmark. We began by researching our expectations for these tools, before evaluating them using the benchmarks. We extensively evaluate and discuss the recall measurements for the tools by each of the benchmarks. We then compared the benchmark results against our expectations and against each other. Additionally, we compared results of our experiment against Bellon's original experiment for CCFinderX, which is a participant in both experiments. We found anomalies in the Bellon's Benchmark results, including significant disagreement with our expectations of the results. Bellon built his corpus by mining the output of tools contemporary to 2002. These clones and Bellon's procedures may not reflect the clone detection and reporting preferences of modern tools. Our findings suggest that Bellon's Benchmark may not be accurate for modern tools, and that an updated corpus may be warranted. In contrast, we did not find anomalies in the measurements of our Mutation and Injection Framework, and found significant agreement with our expectations. The Mutation Framework indicates that ConQat,

iClones, NiCad and SimCad are very good options for detecting all three clone types. We believe the Mutation Framework is be a good solution for benchmarking modern clone detection tools. In particular the Mutation Framework's synthetic benchmarking allows fine-grained measurement of recall in a controlled procedure. However, benchmarking with real data is also important. The results of this study motivated us to create a new real-world clone benchmark, which is the topic of Part II of this thesis.

Chapter 5

Fine-Grained Evaluation with the Mutation and Injection Framework

To demonstrate the our Mutation and Injection Framework for fine-grained recall measurement, we use it to evaluate the recall of ten clone detection tools, including: CCFinderX [58], ConQat [57], Copy Paste Detector (CPD) [99], CtCompare [133], Deckard [53], iClones [41], NiCad [110], SimCad [136], Simian [44] and SourcererCC [116]. We evaluated the tools across six benchmarking experiments, covering all permutations of the two clone granularities (function and block) and three programming languages (Java, C and C#) supported by the Mutation and Injection Framework. We compare these tools and evaluate their strengths and weaknesses at a fine granularity, including per clone type and per mutation operator (i.e., clone edit type from the taxonomy). In these experiments, we demonstrate how fine-grained analysis can reveal new insights into the clone detection tools. In particular, it can identify the capabilities of a clone detection tool, as well diagnose the reasons why a tool's recall may be deficient. Specifically, this is accomplished by comparing a clone detector's recall for different mutation operators, and looking at the consistencies and differences.

This chapter is based on a component of a currently unpublished manuscript entitled "The Mutation and Injection Framework" and authored by myself and Chanchal K. Roy.

This chapter is organized as follows. We discuss our experimental setup including benchmark generation with the Mutation Framework Section 5.1. We discuss the subject tools and their configurations in Section 5.2. We then present and discuss the results in Section 5.3. We summarize the positions of the tools in terms of recall in Section 5.4. We discuss threats to the results in Section 5.5, and conclude this work in Section 5.6.

5.1 Experimental Setup

For these experiments, we used IPScanner (Java), Monit (C) and MonoOSC (C#) as our subject systems. Code fragments for clone synthesis were randomly selected from the following source repositories: JDK6 and Apache-Commons (Java), the Linux Kernel (C), and Mono/MonoDevelop (C#). For each experiment, we configured the framework to select 250 random code fragments from the source repository. We used 15 single-operator mutators, one for each of the 15 mutation operators. Each of these mutators apply a single instance of their assigned mutation operator. From the selected fragments, a total of 3,750 unique clone pairs were synthesized by the mutators. For each clone, we configured the framework to construct 10 mutant

systems using different random injection locations within the subject system. In total, each experiment's reference corpus contains 37,500 mutant subject systems. Across the six experiments, we have constructed 225,000 mutant systems (unique clones) for tool evaluation.

We constrained the generation process to give the reference corpora the following clone properties: (1) 15-200 lines in length, (2) 100-2000 tokens in length, (3) a minimum 70% clone similarity measured by line and by token after Type-1 and Type-2 normalization, and (4) a mutation containment of 15%. Since our experiments contain a large number of mutant systems, we preferred slightly larger clones for our reference corpora. Clone detection tools often run significantly faster when configured for larger minimum clone sizes. This allowed us to evaluate the tools in a reasonable time-frame.

For tool evaluation, we configured the framework's clone matching algorithm to use a subsume tolerance of 15%. This matches the mutation containment, and ensures that a successful match captures the mutant portion of the clone. This is essential for evaluation using mutation analysis, as we want to measure how well the clone detectors handle clones with these particular types of mutations. This is a flexible subsume tolerance that favors the tools in the evaluation.

We measured unit recall using four minimum clone similarity thresholds: 0%, 50%, 60% and 70%. With 0% the framework measures the tool's ability to capture the injected clone, regardless of the quality of the subsuming clone. The non-zero similarity thresholds measure recall with the expectation that the reported clone not only subsumes the injected clone, but contains a minimum degree of syntactical similarity. This is to prevent accepting a match where the candidate clone is suspected to be a buggy clone or a false positive that happens to subsume the reference clone.

We use a range of similarity thresholds as a single threshold may be too rigid. It is difficult to decide which threshold provides the best results. We measure recall for a 0% similarity threshold, to see how well the subject tools capture the reference clones when the quality of the detection is ignored. We use 50% as our weakest definition of a true clone. This requires the code fragments of the true clone pair to share at least half of their syntax, by line or by token. This is a reasonable minimum expectation for a true positive clone of the first three clone types. Our strongest threshold is 70%, which is the minimum similarity of the clones in the generated reference corpora. We also include the 60% threshold as a balance between these two. When we evaluate the tools, we consider how their recall changes as the minimum similarity parameter is varied. This way we overcome the rigidity of a single threshold.

5.2 Participants

The participating subject tools, their programming language and clone type support, as well as their their configurations for the experiments are summarized in Table 5.1. We configured the tools from an experienced user's perspective. An experienced user has explored a tool's configuration options, defaults and documentation, and modifies the tool's settings, where appropriate, for their use case. An experienced user

Table 5.1: Participating Subject Tools, Their Language and Clone Type Support, and Configuration

Tool	Languages	Types	Configuration
CCFinderX	Java, C, C#	1,2	Min length 50 tokens, min token types 12.
ConQat	Java, C	1,2,3	Min length 15 lines, max errors 3, gap ratio 30%.
CPD	Java, C, C#	1,2	Min length 100 tokens, ignore annotations/identifiers/literals, skip parser errors.
CtCompare	Java, C	1,2	Min length 100 tokens, max 3 isomorphic relations.
Deckard	Java, C	1,2,3	Min length 50 tokens, 85% similarity, 2 token stride.
iClones	Java, C	1,2,3	Min length 100 tokens, min block 20 tokens.
NiCad	Java, C, C#	1,2,3	Min length 15 lines, blind identifier normalization, identifier abstraction, min 70% similarity.
SimCad	Java, C, C#	1,2,3	Greedy transformation, unicode support, min 15 lines.
Simian	Java, C, C#	1,2	Min length 15 lines, ignore identifiers and literals.
SourcererCC	Java, C, C#	1,3	Min length 15 lines, min similarity 70%, function granularity.

is not necessarily a clone expert or researcher, but is comfortable configuring the tools for their target input. Specifically, we wanted to measure the performance an experienced user can expect when using these tools with their own subject systems.

To configure the tools we first consulted their documentation and default settings. We enabled any features that provided Type-1 and Type-2 normalization. Clone size and clone similarity thresholds were configured with respect to known properties of the benchmarks. We avoided over-configuring the tools, especially where precision might suffer. We did not execute the experiment for many permutations of a tool's settings to find their optimal configuration for the benchmarks as a tool user would not be able to do this for their own systems.

The tool runners were implemented to execute the tools using these configurations. They then convert the tool's output format to the format expected by the framework. We mentioned previously that the tool runners also receive the clone type of the injected clone, and the mutation operators used to construct it. Since we wanted to evaluate these tools for general clone detection by experienced users, the tool runners ignored this input data. The tools were executed for each mutant system with the same configuration.

Full Type-1 and Type-2 normalizations were not enabled for some tools. CPD only supports these normalizations for Java subject systems. SimCad has two identifier normalization options: systematic renaming and blind renaming. The blind renaming can find more clones, but can hurt precision, so we used systematic renaming. Simian supports identifier normalization for its supported languages, but we found that it produced unusually large clone reports for C# systems with identifier normalization enables. This indicates a bug or a precision problem, so we disabled identifier normalizations with the C# experiments. SourcererCC does not formally support Type-2 clone detection, as it does not use Type-2 normalizations. Instead it targets Type-2 and Type-3 clones using its bag-of-tokens source model and similarity threshold. It does, however, perform stemming on the source tokens which could be seen as a partial Type-2 normalization. We keep these configurations in mind while evaluating these tools.

In a couple cases we did not execute a tool for its supported languages or settings. Deckard's Java parser only supports the Java-1.4 specification, so we did not evaluate it in our Java experiments which use the Java-1.6 specification. While Deckard can still be executed for such systems, its detection performance is compromised by the parser. We only executed ConQat for Java. ConQat is a powerful toolkit for rapid de-

velopment and execution of software quality analysis. Included is functionality for performing clone detection with multiple languages. However, only for Java does it include a pre-configured analysis script for Type-3 clone detection.

5.3 Results

The results of the Mutation and Injection framework benchmark experiment are shown in Tables 5.2, 5.3, and 5.4. These tables summarize the subject tools' recall performances for the three programming languages (Java, C, C#) and two clone granularities (functions, blocks) supported by the framework. Recall is summarized per mutation operator, per clone type. Per clone type statistics were measured by averaging the recall of the respective mutation operators. When comparing the tools' recall across mutation operators or clone granularity, we consider them to be notably different only if they differ by at least 5% (absolute difference). When comparing the recall of the tools, we are using absolute difference of percentage points, not relative difference, unless otherwise stated.

We measured recall by configuring the clone-matching algorithm with four minimum clone similarity thresholds: 0%, 50%, 60%, 70%. To be concise, we only include our recall results for a minimum clone similarity threshold of 60%. Comparing the tools' recall as the threshold was varied, we found 60% to provide the best results.

For most of the tools, recall experienced negligible change when the minimum clone similarity was increased to 60%. A handful of tools had recall drop significantly when the minimum clone similarity was raised to 70%. A 70% threshold may be too strict as this is the same threshold used when generating the reference corpora, and is therefore the target we used to configure the subject tools. Differences between how the framework and the subject tools measure clone similarity may cause some conflict. We therefore prefer the 60% threshold as it is the highest threshold before we see some conflict due to disparate clone similarity metrics. It is a strong enough threshold to reject obvious false positives when measuring unit recall, but not so strong as to be overly strict with the tools.

Only CCFinderX has a noticeably higher recall when the clone-matching algorithm is configured with a minimum clone similarity below 60%. Specifically, CCFinderX's recall for C# functions decreased by ∼10% when the similarity threshold was raised from 0% to 50%, but has negligible difference when raised from 50% to 60%. As a Type-2 clone detector, CCFinderX should not be reporting clones with a similarity less than 100% after Type-1 and Type-2 normalizations, so we can assume these are buggy clones or false positives. The performance of the remaining tools varied negligibly (0-2.5% difference) between 0% and 60% minimum clone similarity threshold.

Table 5.2: Recall Results for Java Function (F) and Block (B) Clones

	CCFX		ConQat		CPD		CtComp.		iClones		NiCad		SimCad		Simian		Sourc.CC	
	F	**B**	**F**	**B**	**F**	**B**	**F**	**B**	**F**	**B**	**F**	**B**	**F**	**B**	**F**	**B**	**F**	**B**
mCC_BT	98	94	91	94	99	98	97	97	100	100	100	100	100	98	91	90	100	100
mCC_EOL	98	94	91	94	99	98	97	98	100	100	100	100	100	98	91	90	100	100
mCF_A	98	94	90	94	98	98	96	98	100	100	100	100	100	98	58	66	100	100
mCF_R	98	94	91	94	98	98	97	98	100	100	100	100	100	98	66	55	100	100
mCW_A	98	94	91	94	99	98	95	98	100	100	100	100	100	98	91	90	100	100
mCW_R	98	94	91	94	99	98	96	98	100	100	100	100	100	98	91	90	100	100
mSRI	91	90	88	92	99	98	95	96	85	82	100	100	100	98	91	90	100	100
mARI	22	20	90	94	99	98	95	96	96	98	100	100	83	89	91	90	100	100
mRL_N	96	93	91	94	0	0	0	0	99	98	100	100	100	97	89	88	100	100
mRL_S	98	94	91	94	99	98	0	0	93	97	100	100	100	97	91	90	100	100
mDL	0	0	85	87	1	1	0	1	94	94	100	100	85	91	0	2	100	100
mIL	0	0	85	87	2	2	0	0	98	97	100	100	90	93	1	1	100	100
mML	0	0	88	90	0	0	0	0	96	97	100	100	83	86	0	0	100	100
mSDL	0	2	86	89	0	0	0	0	95	95	100	100	99	96	0	0	100	100
mSIL	0	2	86	90	0	1	0	1	97	98	100	100	88	87	0	0	100	100
Type-1	98	94	91	94	99	98	96	98	100	100	100	100	100	98	81	80	100	100
Type-2	77	74	90	94	74	74	48	48	93	94	100	100	96	95	90	89	100	100
Type-3	0	1	86	89	0	1	0	0	96	96	100	100	89	91	0	1	100	100

5.3.1 Java

Type-1 Recall

Most of the tools have exceptional Type-1 Java recall. NiCad, iClones, SimCad (functions) and SourcererCC have perfect detection across the mutation operators. CCFinderX (for the function granularity), CPD, CtCompare and SimCad (blocks) have an average Type-1 recall greater than 95%. CCFinderX (blocks) and ConQat have an Type-1 recall greater than 90% but less than 95%. While Simian falls behind the other tools with an average Type-1 recall around 80%. Most of these tools have negligible difference (<5%) between the function and block granularities, with the exception of Simian for Type-1 clones with differences in formatting.

All of the tools, with the exception of Simian, have negligible difference in recall between the mutation operators. When the tools have stable recall across the mutation operators, it suggests the clones they do miss is due to the location of the clone, or the syntax of the clone or its containing source files, not the particular clone differences added by mutation. For example, a tool may have missed a clone due to problems parsing one or both of the files containing the clone.

Only Simian has significantly different recall between the mutation operators. It has poor recall for Type-1 clones with differences in formatting: 58-66% recall. Specifically, it poorly handles the addition or removal of newlines between otherwise identical code fragments. However, it has very good recall for the other Type-1 clone differences: 90-91% recall. This suggests Simian uses some formatting normalization during clone detection, but it is not reliable for all locations where a new-line is valid in a code fragment.

Type-2 Recall

The tools have a wide range of average Type-2 Java recall. NiCad and SourcererCC have perfect recall for both granularities. ConQat, iClones, SimCad and Simian have an average Type-2 recall of ∼90% or greater. CCFinderX and CPD have a lower average recall, around 75%. While CtCompare has a poor average recall, just below 50%. None of these tools have a significant difference in Type-2 recall between the function and block clone granularity.

With the exception of ConQat, NiCad, and Simian, the remaining tools have notable differences in recall for clones with different Type-2 differences (different mutation operators). CCFinderX has very poor recall (20-22%) for clones with an arbitrarily renamed identifier. However, it has very good recall (90-91%) for clones with systematically renamed identifiers. Possibly its partial detection of arbitrarily renamed identifiers is only for cases where the renamed identifier instance was the only instance of that identifier in the clone's code fragments.

CPD has near-perfect recall for clones produced by the Type-2 mutation operators, with the exception of no recall for the mRL_N mutation operator, which changes the value of one numeric literal. While CPD claims to support the normalization of literal values, the results shows it only normalizes string and character literals, not numeric literals.

CtCompare has exceptional recall for clones that contain differences in identifier names, whether systematically or arbitrarily performed. However, it has no recall for Type-2 clones that contain differences in literal values.

iClones has very good recall across all of the mutation operators. However, compared to the other mutation operators, its recall for Type-2 clones with systematically renamed identifiers is lower by 11-16%. However, its detection of clones with arbitrary renamed identifiers is high. In contrast, SimCad has 8-17% lower recall for Type-2 clones with arbitrarily renamed identifiers compared to the other Type-2 mutation operators.

ConQat, NiCad, Simian and SourcererCC have very stable recall across the Type-2 mutation operators. However, ConQat and Simian do not have perfect recall. The uniformity of their per mutation operator recall suggests that they did not fail to detect these clones due to the type of difference added by mutation, but rather due to other factors such as the syntax or location of the Type-2 clone.

Type-3 Recall

Only ConQat, iClones, NiCad, SimCad and SourcererCC support Type-3 Java detection. The framework appropriately measures near-zero Type-3 recall for the strictly Type-2 Java detectors, including: CCFinderX, CPD, CtCompare and Simian. For some Type-3 mutation operators, these Type-2 detectors have a recall of 1-2%. This is likely coincidental. The tool may have detected a Type-1 region of the Type-3 clone, and due to an error in reporting the clone boundary, happened to also capture the Type-3 portion.

Looking at the average Type-3 recall, most of the Type-3 detectors performed quite well. NiCad and

SourcererCC have perfect detection of these Type-3 clones (100%), followed by iClones with an excellent recall of 96%. SimCad has a Type-3 recall around 90%, while ConQat falls just below 90% with a recall of 86-89% depending on the clone granularity.

Overall, there is very little performance differentiation between the clone granularities for these tools. Across the mutation operators, ConQat and Scorpio slightly favor block granularity detection, but the difference is fairly negligible. SimCad shows a small amount of variability across the granularities, with neither granularity being uniformly better. The variability is most pronounced for the mDL mutation operator.

Looking at the per mutation operator performance, ConQat, iClones, NiCad and Scorpio have no or very little difference in recall between the mutation operators. Variance between the mutation operators, within a granularity, is no larger than 4%. This low variance is likely due to how Type-3 clones are detected. While the tools target particular Type-1 and Type-2 differences using normalizations, they do not typically target particular Type-3 differences. Instead they use a uniform clone similarity metric.

When there is variance between the recall of the Type-3 mutation operators, it shows that the different types of Type-3 differences have a non-uniform affect on the similarity metric used. SimCad shows some notable variance across the mutation operators. For both granularities, its detection is strongest for Type-3 clones which differ by the insertion of a line, or a small deletion within a line. Its performance suffers (relative to the other mutation operators) when the Type-3 clones differ by a deletion of a line, or a modification of a whole line. This performance dip is more strong for the function granularity clones. SimCad detects similar code fragments by comparing their Similarity Hash (SimHash). It is possible that the different types of Type-3 differences produce different levels of divergence between the injected code fragment's SimHash values. The benefit of our framework is the SimCad developer could use these results to see how the technique can be modified to improve the detection of the Type-3 clone differences SimCad has lower recall for. Perhaps by modification to the SimHash algorithm with respect to these Type-3 edit types.

5.3.2 C

Type-1 Recall

Most of the tools have exceptional Type-1 recall for C clones. SourcererCC has perfect recall for both function and block granularity. CCFinderX, CPD, NiCad and SimCad have nearly perfect recall for both function and block granularity Type-1 clones. Simian has good recall (82-85%). CtCompare has poor overall Type-1 recall (68-69%), due to no detection of clones that differ by an 'end of line' style comment. Deckard performs poorly with 73% recall for Type-1 function clones, and 59% recall for Type-1 block clones. Deckard is the only tool to have such a significant difference between its detection at the block and function granularities. At the per mutation operator level, Simian, with the mCF_R mutation operator, is the only other tool to have a recall variance larger than 4% between the two clone granularities. The other tools have very similar detection across the clone granularities.

Table 5.3: Recall Results for C Function (F) and Block (B) Clones

	CCFX		CPD		CtComp.		Deckard		iClones		NiCad		SimCad		Simian		Sourc.CC	
	F	**B**	**F**	**B**	**F**	**B**	**F**	**B**	**F**	**B**	**F**	**B**	**F**	**B**	**F**	**B**	**F**	**B**
mCC_BT	100	99	98	99	83	81	73	59	100	100	99	100	100	98	97	93	100	100
mCC_EOL	100	99	98	99	0	0	73	59	100	100	99	100	100	98	99	97	100	100
mCF_A	100	99	96	99	83	81	73	59	100	100	99	100	100	98	63	61	100	100
mCF_R	100	99	95	99	83	81	72	58	100	100	99	99	99	97	59	53	100	100
mCW_A	100	99	98	99	83	81	73	59	100	100	99	100	100	98	96	94	100	100
mCW_R	100	99	98	99	83	81	73	59	100	100	99	100	100	98	96	95	100	100
mSRI	98	98	0	0	80	81	72	59	90	95	99	100	98	97	99	97	100	100
mARI	33	30	0	1	80	81	73	59	99	99	99	100	92	93	99	97	100	100
mRL_N	100	99	0	0	0	0	73	59	98	99	99	100	100	98	90	85	100	100
mRL_S	100	99	0	0	0	0	73	59	98	96	99	100	100	98	99	97	100	100
mDL	0	0	0	0	0	0	66	54	98	97	99	99	89	88	0	0	100	100
mIL	0	0	0	1	0	1	65	56	99	98	99	100	96	92	0	1	100	100
mML	0	1	0	1	0	1	72	58	99	97	99	100	81	86	0	0	100	100
mSDL	0	0	0	0	0	0	72	58	98	98	99	100	96	95	0	0	100	100
mSIL	0	1	1	1	1	1	72	59	99	98	99	100	84	89	0	0	100	100
Type-1	100	99	98	99	69	68	73	59	100	100	99	99	100	97	85	82	100	100
Type-2	83	82	0	0	40	40	73	59	96	97	99	100	97	96	97	94	100	100
Type-3	0	0	0	1	0	1	69	57	99	98	99	100	89	90	0	0	100	100

CtCompare and Simian are the only tools to have a notable difference in recall across the Type-1 mutation operators. CtCompare has good performance (81-83%) for most of the Type-1 mutation operators, but does not detect any of the Type-1 clones that differ by a end-of-line style comment (e.g., '//comment'). CtCompare does not display this weakness for Java clones produced by the mCC_EOL mutation operators, which hints that there is a bug in CtCompare's C syntax parser or normalizer.

Simian has excellent (90-91%) recall for all of the Type-1 mutation operators, but poor recall (55-66%) for clones with differences in formatting (the mCF_A and mCF_R mutation operators). Simian is a line-based tool, and it seems its normalization for differences in formatting does not work for all cases.

Deckard generally has low recall for the Type-1 clones, but its recall is very uniform across the Type-1 mutation operators. This suggests that Deckard does not struggle with any particular Type-1 difference, but rather it is struggling with the syntax of the clones, or their injection locations (i.e., the surrounding source code).

Type-2 Recall

Many of the tools have exceptional Type-2 recall for C clones of block and function granularity. SourcererCC has perfect Type-2 detection for both granularities, while NiCad has perfect detection for the block granularity and near-perfect (99%) for the function granularity. iClones, SimCad and Simian have an average Type-2 recall greater than 90%. CCFinderX has a good average Type-2 recall just over 80%. Deckard has poorer performance with 73% Type-2 recall for function clones, and 59% for block clones. CtCompare has very poor performance with an average of 40% recall. CPD has 0% recall for Type-2 clones because it does not support

Type-2 normalizations with C.

Most of the tools have uniform recall across the clone granularities. Simian for the mRL_N mutation operator, and iClones for the mSRI mutation operator, have a variance of 5% between the clone granularities. Only Deckard has a consistently different recall for function and block clones. Deckard has a respectable recall for Type-2 function clones (73%) but a poor recall for Type-2 block clones (59%). Otherwise, these tools have similar recall between the granularities.

A number of tools have weaknesses for particular Type-2 clone differences. CCFinderX has very poor recall (30-33%) for Type-2 clones that differ by arbitrary renaming of identifiers (mARI), despite near-perfect recall for Type-2 clones produced by the other mutation operators. CtCompare has good (80-81%) recall for Type-2 clones that differ by identifier names, but has no recall for clones that differ by literal values; it must not support these normalizations. iClones has near-perfect (93-99%) recall for all of the Type-2 mutation operators except for somewhat lower (82-85%) recall for clones produced by the mSRI mutation operator. This is for clones with a systematically renamed identifier. Since iClones has near-perfect recall for clones with arbitrarily renamed single identifier instances, it could be that it struggles when the number of renamed instances of an identifier is large. SimCad has slightly lower recall for clones with arbitrarily renamed identifiers. This is likely due to SimCad being configured to systematically normalize identifiers names, because its arbitrary normalization can cause false positives. Simian, particularly for the block granularity, has lowered performance (∼10% drop) for Type-2 clones with differences in numeric literal values. This is odd because it has near-perfect detection for Type-2 clones with differences in string literal values.

Type-3 Recall

The framework correctly measures a near-zero recall for the tools that do not support Type-3 detection: CCFinderX, CPD, CtCompare and Simian. The detection of Type-3 C clones is supported by Deckard, iClones, NiCad, SimCad and SourcererCC. SourcererCC has perfect detection, while iClones and NiCad has nearly-perfect detection (98% and higher). SimCad also has strong detection, averaging around 90%. Deckard has poor performance that differs slightly between the clone granularities. It has a 69% recall for function clones, and 57% recall for block clones. The other tools have only negligible variance between the clone granularities.

NiCad and iClones have negligible variance in recall across the Type-3 mutation operators. Deckard performs worse for the deletion (mDL) and insertion (mIL) of lines in Type-3 clones compared to the other mutation operators, but this variance is only significant for the function granularity clones. SimCad has notable variability in performance across all of the Type-3 mutation operators, ranging from 81-96% recall. While its performance for the mutation operators ranges from good to excellent, some tweaks to its SimHash similarity function might reduce this variance. This shows how valuable the Mutation Framework is for measuring fine grained clone detector performance for tool improvement. It can be used to spot the specific detection cases a tool could be improved for.

5.3.3 C#

Type-1 Recall

NiCad, SimCad and SourcererCC have nearly perfect recall for the Type-1 clones C# clones. CCFinderX also has good recall, with 90% for the function granularity and 96% for the block granularity. CCFinderX is the only tool to have a variance in recall of 5% or greater between the granularities. Simian has mediocre performance, with an average of 75% for the function granularity, and 73% for the block granularity. This is caused by its poor detection of clones that contain differences in formatting. Its recall for the mCF_A and mCF_R is only 49-53%. Its recall for the other Type-1 mutation operators is good, with a steady 84% for the block granularity, and 88% for the function granularity. Simian is the only tool to show a notable variance across the Type-1 mutation operators. CPD only detects approximately half of the Type-1 clones. However, its recall is very uniform across both the mutation operators and granularities. Possibly CPD is having difficulty handling the C# syntax of the injected clones, or the source files they were injected into.

Type-2 Recall

SourcererCC has perfect detection for Type-2 C# clones, while NiCad has near-perfect recall for all Type-2 mutation operators (98-99%). SimCad also has strong detection, 97-100% for most mutation operators, with some weakness in the detection of Type-2 C# clones with an arbitrarily renamed identifier. This is due to SimCad being configured to systemically rename identifiers, rather than arbitrary renaming, as otherwise its precision can suffer. It still has good recall for the mARI produced clones despite this limitation. CCFinderX has weaker performance, with an average 75% recall for Type-2 function clones, and an average 80% for Type-2 block clones. Particularly, CCFinderX has very poor performance for clones with arbitrarily renamed identifiers. It has good to excellent performance for the other mutation operators. Other than with the mARI mutation operator, CCFinderX has stronger recall (by 6%) for the block granularity.

Simian has good (82-85%) performance for Type-2 clones with differences in literal values. It has no detection of the Type-2 clones with differences in identifiers because this normalization was disabled for the C# experiments. Enabling this normalization caused Simian to produce very large detection reports despite the smaller size of the C# subject system. It seems this normalization may harm Simian's precision for C#, or causes errors in clone reporting. The Mutation Framework correctly identifies CPD as not supporting Type-2 normalizations in C# systems.

Type-3 Recall

Only NiCad, SimCad and SourcererCC support the detection of Type-3 C# clones. The framework correctly identifies that CCFinderX, CPD, and Simian do not support Type-3 detection, with near-zero recall for the Type-3 mutation operators. SoucererCC has very nearly perfect recall, with 99% recall for the mIL mutation operator at the block granularity. NiCad has almost perfect Type-3 detection, with per mutation operator

Table 5.4: Recall Results for C# Function (F) and Block (B) Clone Pairs

	CCFX		CPD		NiCad		SimCad		Simian		Sourc.CC	
	F	B	F	B	F	B	F	B	F	B	F	B
mCC_BT	90	96	51	50	98	99	100	97	88	84	100	100
mCC_EOL	90	96	52	51	98	99	100	97	88	84	100	100
mCF_A	90	96	51	51	98	99	100	97	51	53	100	100
mCF_R	89	96	52	51	98	98	100	96	50	49	100	100
mCW_A	90	96	51	50	98	99	100	97	88	84	100	98
mCW_R	90	96	52	51	98	99	100	97	88	84	100	100
mSRI	88	94	0	0	98	99	100	97	0	0	100	100
mARI	33	35	0	0	98	99	88	85	0	0	100	100
mRL_N	88	94	0	0	98	99	100	97	86	83	100	100
mRL_S	90	96	0	0	98	99	100	97	88	82	100	100
mDL	0	2	1	0	97	98	89	83	1	1	100	100
mIL	0	0	0	0	98	99	93	90	0	0	100	99
mML	0	0	0	0	98	99	78	83	0	0	100	100
mSDL	0	0	0	0	98	99	98	94	0	1	100	100
mSIL	0	0	0	0	98	99	82	82	1	0	100	100
Type-1	90	96	52	51	98	99	100	97	75	73	100	99
Type-2	75	80	0	0	98	99	97	94	43	41	100	100
Type-3	0	0	0	0	98	99	88	86	0	0	100	100

recall ranging from 97% to 99%. NiCad has negligible variance in its recall across the mutation operators and clone granularities. SimCad has good Type-3 detection, with an average Type-3 recall of 88% for function clones and 86% for block clones. SimCad has notable variation in recall across the clone granularities for the mDL and mML mutation operators. It also shows variance across the mutation operators. It performs best for clones produced by the mSDL and mIL mutation operators, and worst for those produced by mML.

5.4 Summary

We explored the performance of these tools per language and clone type. From these results, we can comment on the overall performance of these tools and make some recommendations. The iClones, NiCad and SourcererCC clone detectors are perhaps the best contenders, with excellent performance across the languages, granularities and clone types. The experiments did not find any particular weaknesses in NiCad or SourcererCC. While iClones has very strong overall performance, it does not support C#, and has a drop in recall for Type-2 clones with systematically renamed identifiers.

SimCad also has strong detection across the three clone types and languages, but has a couple weaknesses. In order to maintain precision, its identifier normalization must be applied systematically. This causes it to miss some Type-2 clones with arbitrarily renamed identifiers (∼80% recall). It also has a high degree of variance in its Type-3 detection, ranging from 78-99%.

ConQat has good Java recall, with ∼90% for Type-1 and Type-2 clones, and 85-90% for Type-3 clones. ConQat is a general framework for software analysis. While it appears to support clone detection in other languages, it only contained a prepared script for Type-3 detection for Java. Deckard has rather poor recall

for C clones, with an average of 71% for function clones, and 58% for block clones. While Deckard can be executed for Java systems, it only supports source files that conform to the Java-1.4 specification, which our benchmark contains code up to the Java-1.6 specification.

While the Type-2 clone detectors have excellent detection in some cases, they all have a weakness in their detection. CCFinderX supports all three languages, and has excellent overall Type-1 and Type-2 detection. However, its detection is very poor for tools with arbitrary renamed identifiers. CPD has very good Type-1 detection for Java and C, but very poor detection with C#. CPD supports Type-2 clone detection with Java only. It has excellent Type-2 Java recall, except for clones that contain differences in numeric literals, for which it has no recall. CtCompare has high (90%) recall with Java, and good (80%) recall with C, for the types of Type-1 and Type-2 clone differences it supports. However, it does not detect any Type-2 clones with differences in literal values, with either language. Nor does it detect Type-1 C clones that contain differences in end of line style comments (e.g., '//comment'). Simian has good overall Type-1 and Type-2 detection for Java and C. However, it has weak performance (50-66%) for clones that differ by formatting.

Overall, with the exception of Deckard, we did not find significant differences in recall between the function and block granularity. The block granularity results are best for measuring the general clone detection recall of these tools. Function clone detection is important as this is the granularity a developer may first examine when analyzing the clones in their software system, as function clones can be easier to understand and refactor than block clones. Functions encapsulate complete logical functionalities, whereas blocks may encapsulate only a part of a functionality, and therefore the entire function may need to still be considered.

Many of the tools had a weakness in detection clones produced by at least one of the mutation operators (editing activities of the taxonomy). In most cases the weakness was seen in all of the supported languages. Although in a few cases a tool supported the detection of a particular kind of clone in one language but not another. For example, CtCompare could not detect Type-1 clones that differ by an end-of-line style comment in C, but did not have this weakness with Java.

Many of the tools have particular weaknesses. The advantage of using the Mutation and Injection Framework is it can isolate these weaknesses at a fine granularity. The results can be used by tool developers to determine aspects of their tool that need improvement. Tool users can use this information to decide which tool they should use for their task. Users can overcome weaknesses in an individual tool by using multiple tools and merging their results. This allows the user to benefit from the unique detection characteristics of multiple tools.

5.5 Threats to Validity

Alternate tool configurations may result in better or worse performance in the tool evaluations. This is referred to as the confounding configuration choice problem by Wang et al. [139]. We took steps to ensure the tool configurations were appropriate for our study. We used configurations that target the known properties

of the benchmark corpora, such as clone types, size and similarity thresholds. We consulted the default settings and documentation of the tools to choose these configurations. We were careful not to configure the tools in a way that would boost recall at a significant reduction in precision. This is the process an experienced user would use to configure these tools for their own subject system. Therefore our results reflect what a user can expect from these tools.

5.6 Conclusion

In this chapter, we used the Mutation and Injection Framework to perform fine-grained recall analysis of ten clone detection tools. We measured recall per clone edit type for function and block clones for the Java, C and C# programming languages. By comparing a clone detection tool's recall across the mutation operators and granularities, we were able to analyze their capabilities at a finger granularity than possible with a real-world benchmark. This chapter demonstrates the true power of synthetic clone benchmarking.

Chapter 6

Evaluating Clone Detection Tools for Gapped Clones

The Mutation and Injection Framework was designed to be extensible by allowing the user plug-in custom mutation operators. The user can design novel mutation operators and combine them in mutators to synthesize any kinds of clones for recall evaluations. For example, if a researcher designs a new clone detector to improve detection of a certain kind or paradigm of clone, perhaps even more specific than one of the clone types, they can customize the Mutation Framework to randomly generate thousands reference clones of the kind they are targeting for rigorous evaluation and comparison against other tools. Otherwise, it can be difficult to demonstrate and evaluate clone detectors that target specific types or novel kinds of clones, which may not be featured in existing clone benchmark datasets.

As a demonstration of this extensibility, we use our Mutation and Injection Framework to rigorously evaluate clone detection tools for Type-3 clones with a single dissimilar gap. This is the case where a code fragment has been copy and pasted, and a number of new sequential source lines (statements) have been inserted into the copied version. This is a very specific sub-type of a Type-3 clone, although a potentially interesting one.

An example of a single gap clone is shown in Figure 6.1. A method was implemented to encapsulate the file copying logic, including a logging message. This was later copied, and the copy was evolved by adding logic to throw an exception if the source file does not exist, or the destination file already exists. The code fragments are identical except for the single gap inserted into the copied version. This is potentially a dangerous clone, as the purpose of the original function may have been to unify the file copying logic used in the software system. Having two versions creates confusion and may cause bugs when developers expect consistent behavior. It is desirable to detect clones like this one. However, this clone is not detectable by most clone detectors as the code fragments are only 50% similar by line, whereas a 70% threshold is standard in clone detection. Existing clone detectors may only be good at detecting single gap clones with small gaps.

Therefore we are interested in how robust the clone detection tools are against Type-3 clones with a single dissimilar gap of any size. In Chapter 5 we evaluated the tools for Type-3 clones with added statements, but only when the gap was a single line. Here we design mutation operators to create Type-3 clones with a single gap of variable lengths. We evaluate the recall of the clone detection tools for single gapped clones with gap size of one to twenty lines.

This chapter is organized as follows. In Section 6.1 we discuss the experimental setup, including the

```
// Original:
public copyFile(Path src, Path copy) throws IOException {
    Files.copy(src,dest);
    Logging.getLogger().log("File " + src + " copied to " + copy);
}


// Cloned with added gap:
public copyFile(Path src, Path copy) throws IOException {
    if(!Files.exists(src))
        throw new IllegalArgumentException("Source does not exist");
    if(Files.exists(copy))
        throw new IllegalArgumentException("Destination already exists!");
    Files.copy(src, dest);
    Logging.getLogger().log("File " + src + " copied to " + copy);
}
```

Figure 6.1: Example Single Gap Clone

creation of the mutation operators, and the setup of the Mutation Framework for generating a single gap clone corpus. In Section 6.2 we discuss the participating tools and their configuration. We discuss the results in Section 6.3 and conclude this chapter in Section 6.4.

6.1 Experimental Setup

6.1.1 Gap Mutation Operator

We created a clone-producing mutation operator for creating Type-3 clones with a single dissimilar gap. This operator takes a code fragment as input, and outputs a mutant version with a single dissimilar gap inserted after a randomly selected line. The operator is configured with the size of a gap to construct. The mutation operator has a database of source lines taken from a collection of software systems which it selects from randomly to create the gap. For example, for a gap size of five lines, the operator will select five random statements from its database and insert these in sequence after a randomly chosen line in the input code fragment.

6.1.2 Corpora Synthesis

We configured the Mutation Framework with twenty mutators using our mutation operator. Each mutator used a single instance of the operator configured to inject a gap of size one to twenty (each mutator produces gap clones of a certain gap length). We used IPScanner (Java) as our subject system, and a combination of JDK6 and Apache Commons libraries as our source repository. We configured the framework to randomly select 200 code fragments from the source repository and produce a gap clone using each of the mutators.

This means we synthesized 200 clones per gap length of one to twenty lines for a total of 2000 synthetic gap clones.

We constrained clone synthesis to clones that are 15-100 lines in length, and 50-1000 tokens in length. We did not enforce a mutation containment so the gap could occur anywhere within the clone. We also did not enforce a minimum syntactical similarity on the synthesized clones, as we want to also generate true clones with large gaps that cause overall syntactical similarity to become low, and therefore be difficult to detect. In this way we will see when the tools encounter difficulty in detecting gapped clones due to the affect a large gap can have on overall syntactical similarity.

6.1.3 Evaluation

For tool evaluation, we configured the framework's clone matching algorithm to use a subsume tolerance of one line to ignore any "off by one line" errors, which are common in clone detection [13]. We measured recall using a minimum required clone similarity of 0%. Since we are generating clones that may have low syntactical similarity, we cannot put this extra constraint on the matching requirement, as we were able to previously (Chapter 5).

6.2 Participants

The participating tools and their configurations for this experiment are summarized in Table 6.1. Since we are producing Type-3 clones, we include only clone detectors that support Type-3 clone detection in this study. Our interest is to see how their detection performance stands up as the gap length increases.

6.3 Results

The recall of the participating clone detection tools is plotted against gap size in Figure 6.2. For all of the participating tools, recall falls as the gap size increases. This is due to a large gap size having a higher chance of dropping the clone's syntactical similarity below the tools' minimum similarity configurations.

NiCad has the strongest recall, with perfect detection of the clones with a gap size of one line, and remaining above 90% until a gap length of four lines. Afterwards, NiCad's recall quickly drops as gap size increases. SourcererCC has good recall for gap lengths of one to two lines, but drops quickly after that.

Table 6.1: Participating Tools and their Configuration

Tool	Configuration
Deckard [53]	Min Length 50 tokens, 85% similarity, 2 token stride.
iClones [41]	Min length 50 tokens, min block 20 tokens.
NiCad [110]	Min length 10 lines, blind identifier normalization, literal abstraction, min 70% similarity.
SourcererCC [116]	Min length 15 lines, min similarity 70%.

iClones has poor recall even for clones with a single gap line, and recall quickly drops off after this. This is because iClones detects Type-1 and Type-2 clones and merges those that are separated by only a small gap, so it is not robust against large gaps. While we previously found iClones to have good recall for Type-3 clones with a single-line gap (Chapter 5), here we are using a newer version of iClones which appears to have introduced some regression in Type-3 detection.

Deckard in particular has poor recall even for the single gap line clones. However, Deckard's parser has some limitations for Java syntax, which could be hurting its performance.

What we are seeing is that the state of the art tools struggle to detect Type-3 clones that have low overall syntactical similarity due to a large dissimilar gap. The clones we synthesized are just simple Type-3 clones, where a code fragment has been copied and pasted and a section of new code added, so there is no doubt they are true clones. This shows that there is a need for a specialized clone detector that can find these clones. While existing clone detectors could find them by lowering their thresholds, this would also reduce overall precision. Possibly a specialized detector could maintain precision by targeting the detection of just the simple Type-3 clones with a large gap.

6.4 Conclusion

In this chapter we demonstrated how the Mutation Framework can be extended with custom mutation operators to evaluate clone detection tools for very specific kinds of clones. For our demonstration, we targeted simple single gap Type-3 clones, where a code fragment has been copy and pasted and a single gap created by the addition of one or more sequential statements within the cloned fragment. We showed that even the best of the state of the art tools have trouble detecting clones with gaps larger than five lines (statements) due to the gap causing low overall syntactical similarity, which the clone detectors rely on for detection. Despite the simplicity of these clones, they are problematic to detect. A specialized clone detector is perhaps needed to target these larger gap clones, and the Mutation Framework can be used to motivate and test such a new clone detector.

We do not focus on larger gap clones further in this thesis. Our goal here was to perform a case study that demonstrates the value of the Mutation Framework's extensibility. The Mutation Framework can be easily extended with new mutation operators to motivate and rigorously evaluate emerging specialized clone detectors.

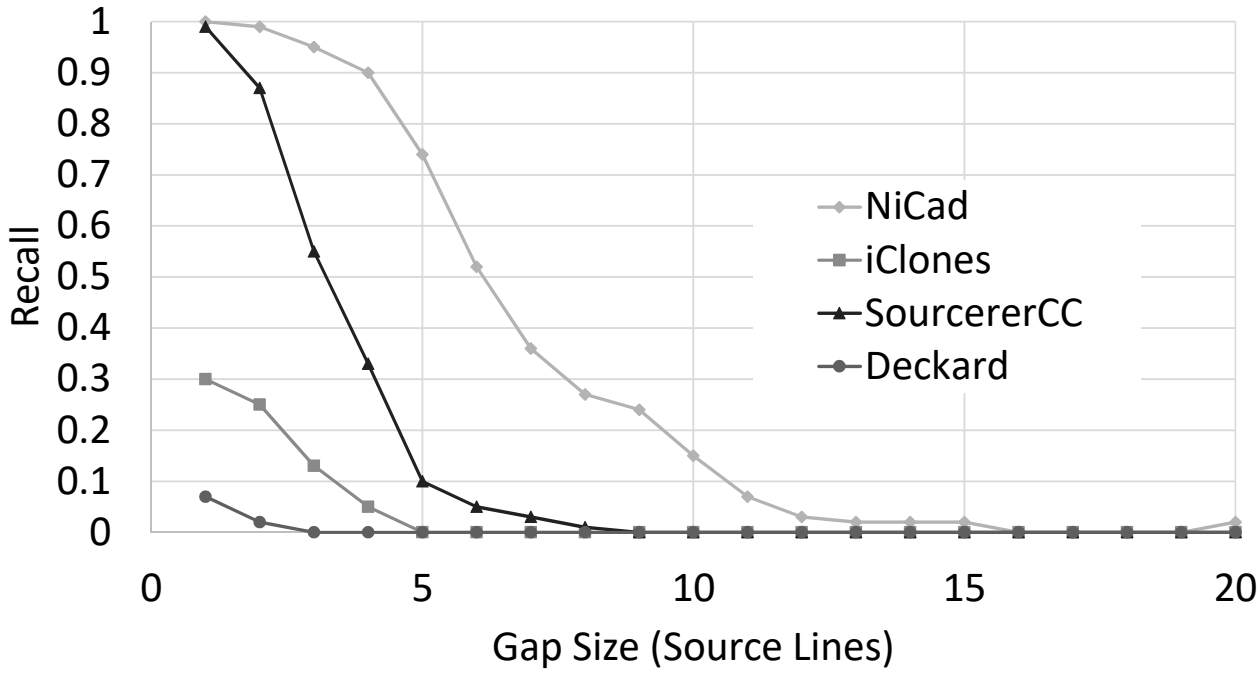


Figure 6.2: Recall Results

Chapter 7

ForkSim: Generating Software Forks for Cross-Project Analysis Benchmarking

In software development, similar software projects called software variants can emerge in various ways. Although systematic reuse approaches such as software product lines are known to enable considerable effort savings [23], existing projects are frequently forked and modified to meet the needs of particular clients and users [31]. These variants typically undergo further parallel development and evolution, and reuse techniques are often not explored until after the variants have matured. This leads to an increased maintenance effort as many tasks are duplicated across the variants.

Maintenance effort can be reduced by merging the forks or by adopting a software reuse approach (e.g., software product lines). Berger et al. [16] report that 50% of industrial software product lines developed by participants of their study were created in an extractive way, i.e., by merging already existing products. This indicates a substantial practical demand for cross-project similarity detection approaches that help software developers discover the similarities between their software variants and support decisions on reuse adoption strategy. Several such approaches have been proposed (e.g., [33,90]), while clone detectors could also be used for this purpose.

A current need is a benchmark for evaluating the performance of tools which detect similarity between software variants. Performance is measured in terms of recall and precision. Recall is the ratio of the similar source code shared between the variants that the tool is able to detect and report. Precision is the ratio of the similar source code elements reported by the tool which are not false positives.

Measuring recall and precision involves executing the tool for a benchmark dataset (or datasets) and analysing the tool's output. Precision can be easily measured by manually validating all (or typically a random subset) of the tool's output. However, measuring recall requires that all similar code amongst the variants of the dataset be foreknown. This makes it very difficult to use datasets from industry or open source (e.g., BSD Unix forks). Building an oracle by manually investigating the dataset for similar code is, due to time required, essentially impossible for datasets large enough to allow meaningful performance evaluation.

To address these difficulties, and to reduce the amount of required manual validation to a minimum, we developed ForkSim, which uses source code mutation and injection to construct datasets of synthetic software forks. The forks are generated such that their similarities and differences at the source level are known. Recall

can then be measured automatically by procedurally comparing a tool's output against the dataset's known similarities. Precision can be measured semi-automatically, as reported similarities which match known similarities or differences in the dataset can be automatically judged as true or false positives, respectively. Only the reported similar code not matching known properties needs to be manually investigated.

The forks generated by ForkSim can be used in any research on detecting, visualizing, or understanding code similarity among software products. The generated forks are a good basis for evaluating automated analysis approaches, as well as for performing controlled experiments to investigate how well the specific tool supports users in understanding similar code. ForkSim is publicly available for download[1].

This chapter is based upon our manuscript [124] "ForkSim: Generating Software Forks for Evaluating Cross-Project Similarity Analysis Tool" published by myself, Chanchal K. Roy and Slawomir Duszynski in the tool paper track of the Working Conference on Software Code Analysis and Manipulation, ©2013 IEEE. I was the lead author of this paper and study, under the supervision of my supervisor Chanchal K. Roy and Slawomir Duszynski. The publication has been re-formatted for this thesis, with small modifications to better fit the thesis.

The remainder of this chapter is organized as follows. Related work is discussed in Section 7.1, and software forking in Section 7.2. Section 7.3 outlines ForkSim's fork generation process, Section 7.4 outlines the comprehensiveness of the simulation, and Section 7.5 discusses the quality of the generated forks. Section 7.6 outlines ForkSim's use cases, while Section 7.7 provides a demonstration of its primary use case: tool performance evaluation. Finally, Section 7.8 concludes the paper.

7.1 Related and Previous Work

Although automatic tool benchmark construction has been proposed in various problem domains [85], to the best of our knowledge there are no other tools which generate software fork datasets for evaluating cross-project similarity analysis tools, nor is there a reference case (e.g., a set of software forks with known properties) which could be used for tool evaluation. The most related work to ours is a manual validation of cross-project similarity analysis results obtained through clone detection by Mende et al. [90]. However, manual result validation has several drawbacks, as discussed in the introduction. ForkSim is unique in that tool evaluation can be mostly automated for datasets generated by ForkSim, as the similarities and differences between the generated software forks are known.

7.2 Software Forking

In the forking process, a software system is cloned and the individual forks are evolved in parallel. Development activities may be unique to an individual fork, or shared amongst multiple forks. For example,

[1] `http://homepage.usask.ca/~jes518/forksim.html`

code may be copied from one fork to another. While forks may share source code, the code may contain fork-specific modifications, and may be positioned differently within the individual forks. A fork may itself be forked into additional forks. Table 7.1 provides a taxonomy of the types of source code level development activities performed on forks. These development activities describe how a fork may change with respect to its state at the start of the forking process. This taxonomy is based upon our research and development experience, and discussions with software developers.

Existing code originates from pre-fork development, and new code originates from post-fork development. The development activities occur at various code granularities, including: source directory, source file, function, etc. The result of forking and these further development activities are a set of software variants containing source code in the following three categories: (1) code shared amongst all the forks, (2) code shared amongst a proper subset of the forks, and (3) code unique to a specific fork. ForkSim creates datasets of forks resulting from these development activities, and containing source code in these three categories. It does this by simulating a simple forking scenario.

7.3 Fork Generation

ForkSim's generation process begins with a base subject system, which is duplicated (forked) a specified number of times. Continued development of the individual forks is simulated by repeatedly injecting source code into the forks. Specifically, ForkSim injects a user specified number of functions, source files, and source directories. Instances of source code of these types are mined from a repository of software systems, which ensures the injected code is realistic and varied.

Each of the chosen functions, files and directories are injected into one or more of the forks. The number and particular forks to inject a source artifact into are selected at random. Injection into a single fork creates code unique to that fork, while injection into a subset of the forks creates code shared amongst those forks. Injection locations are selected randomly, but only amongst the code inherited from the base system, i.e., not inside previously injected code. This prevents the injected code from interacting, which makes the generation process much easier to track and thereby simplifies tool evaluation using the generated dataset. When code is injected into multiple forks, the injection location may be kept uniform or varied, given a specified probability. Forks may share code, but that code may be positioned differently within the individual forks.

Table 7.1: Taxonomy of Fork Development Activities

ID	Development Activity
DA1	New source code is added.
DA2	Existing source code is removed.
DA3	Existing source code is modified and/or evolved.
DA4	Existing source code is moved.
DA5	Source code is copied from another fork. It may be copied into a different position than in the source fork, and it may be modified and/or evolved independently of the source.
DA6	A fork may itself be forked.

Before code is injected into a fork it may be mutated, i.e., automatically modified in a defined way, given a specified probability. This causes code shared by the forks to contain differences, simulating that shared code may be modified or evolved independently for the needs of a particular fork.

Files and directories may be renamed before injection, given a specified probability. While forks may share code at the file and directory level, they may not have the same name. For example, they may have been renamed to match conventions used in the fork.

Each injection operation (injection of a function, a file, or a directory) is logged. This includes recording the forks the code was injected into, the injection locations used, and if and how the code was mutated and/or renamed before injection. A copy of the code and any of its mutants are kept separately and referenced by the log. ForkSim also maintains a copy of the original subject system. From the log and the referenced data the directory, file and function code similarities and differences inherited from the original system and introduced by injection, can be procedurally deduced.

Fig. 7.1 depicts this generation process. On the left side are the inputs: the subject system, source repository and user-specified generation parameters. The subject system duplicates (forks) are modified by the boxed process, which repeats for each of the source files, directories and functions to be introduced. The figure shows an example of this process, in which a randomly selected function from the source repository is introduced into the forks. ForkSim randomly decided to inject this function into three of the four forks using non-uniform injection locations, and to mutate the function before injection into the latter two forks. On the right side are the outputs: the generated fork dataset and the generation log.

ForkSim supports the generation of Java, C and C# fork datasets. It is implemented in Java 7 (main architecture and simulation logic) and TXL [27] (source code analysis, extraction and mutation). ForkSim's generation parameters are summarized in Table 7.2.

Injection Directory and file injection is accomplished by copying the directory or file into a randomly selected directory in the fork. In order to prevent injected directories from dominating a fork's directory tree, only leaf directories (those containing no subdirectories) are selected for injection. Function injection is performed by selecting a random source file from the fork, and copying the function's content directly after a randomly selected function in the chosen file. The generation process can be parametrized to only select functions for injection which fall within a specified size range measured in lines before mutation.

Mutation ForkSim uses mutation operators to mutate code before injection. Fifteen mutation operators, named and described in Table 7.3, were implemented in TXL [27]. Each operator applies a single random modification of its defined type to input code. These mutation operators are based upon a comprehensive taxonomy of the types of edits developers make on cloned code [108], which makes them suitable for simulating how developers modify shared code duplicated between forks.

Files and functions are mutated by applying one of the mutation operators a random number of times before injection. The number of mutations is limited to a specified ratio of the size of the file/function

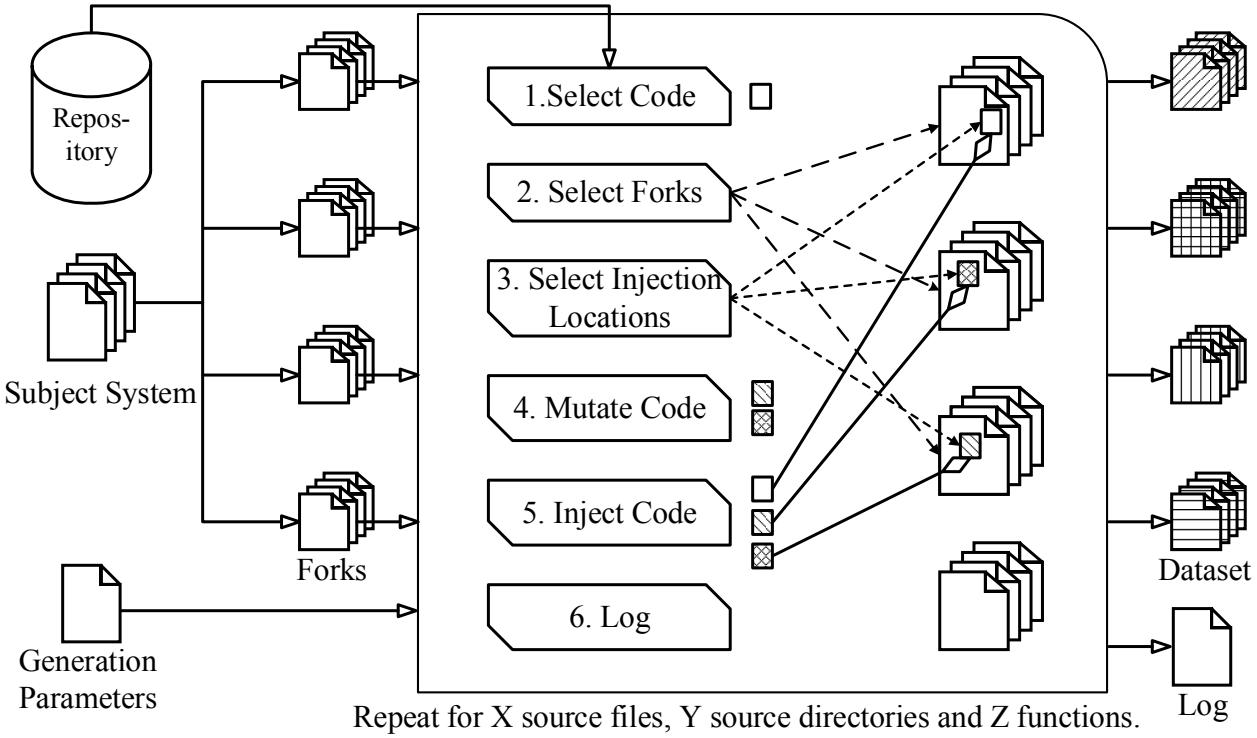


Figure 7.1: Fork Generation Process

measured in lines after pretty printing. This provides an upper limit on how much simulated development is allowed to occur on a source file or function in a particular fork. Pretty printing (one statement per line, no empty lines, comments removed) the source artifact before measurement ensures the measure is consistently proportional to the amount of actual source code contained. This ratio can be specified separately for files and functions.

A small mutation ratio is recommended (10-15%) as too many changes may cause the variants of a file/function injected into multiple forks to become so dissimilar that they may no longer be a clone. Detection tools would be correct not to report them. ForkSim datasets are only useful if the elements declared as similar are indeed similar.

When directories are injected, each of the source files in the directory may be mutated using the same process as used for injected files. The directory mutation probability parameter defines how likely a file in an injected directory is mutated.

As a principle of mutation analysis, ForkSim does not mix mutation operators. This makes it easier to discover if a similarity analysis tool struggles to detect similar code with particular types of differences. ForkSim cycles through the mutation operators to ensure that each is represented in the generated forks roughly evenly. When it is not possible to apply a given operator to the file or function, another operator is chosen randomly.

Table 7.2: ForkSim Generation Parameters

Parameter	Description
Subject System	The base system which is forked during the generation process.
Source Repository	A collection of systems from which the source files, directories and functions are mined.
Language	Language of forks to generate (Java, C or C#).
# Forks	Number of forks to generate.
# Files	Number of files to inject.
# Directories	Number of directories to inject.
# Functions	Number of functions to inject.
Function Size	Maximum/Minimum size of functions to inject.
Max Injections	Maximum number of forks to inject a particular function/file/directory into.
Uniform Injection Rate	Probability of uniform injection of a source artifact.
Mutation Rate	Probability of source mutation before injection. Specified separately for function, file and directory injections.
Rename Rate	Probability of renaming before injection.
Max Mutations	Maximum number of mutations to apply to injected code. Specified as a ratio of the code's size in lines.

Table 7.3: Mutation Operators from a Code Editing Taxonomy for Cloning

ID	Description
mCW_A	Change in whitespace (addition).
mCW_R	Change in whitespace (removal).
mCC_BT	Change in between token (/* */) comments.
mCC_EOL	Change in end of line (//) comments.
mCF_A	Change in formatting (addition of a newline).
mCF_R	Change in formatting (removal of a newline).
mSRI	Systematic renaming of an identifier.
mARI	Arbitrary renaming of a single identifier.
mRL_N	Change in value of a single numeric literal.
mRL_S	Change in value of a single string literal.
mSIL	Small insertion within a line (function parameter).
mSDL	Small deletion within a line (function parameter).
mIL	Insertion of a line.
mDL	Deletion of a line.
mML	Modification of a whole line.

Renaming The probability of a file or directory being renamed before injection is specified separately from that of source code mutation, and both are allowed to occur on the same injection. Renamed source files keep their original extensions.

Usage ForkSim operation is very simple. The user makes a copy of the default generation parameters file, and tweaks it for the dataset they wish to generate. This includes specifying paths to the subject system and source repository to use. Once the parameters file and systems are prepared, the user executes ForkSim and specifies the location of the parameters file and a directory to output the forks and generation log into. Once execution is complete, the forks and generation log are ready for use in experiments involving similarity analysis tools.

7.4 Simulation of Development Activities

During the generation process, ForkSim simulates all six of the development activities from the forking taxonomy (Table 7.1). The following subsections describe how the code injection scenarios performed during the generation process can be interpreted as the six development activities.

DA1 Any of the code injections can be interpreted as the addition of new code to a fork.

DA2 Code injected into a proper subset of the forks can be interpreted as existing code (pre-fork) which was deleted from the forks it was not injected into.

DA3 Code injected into the forks, with at least one instance mutated, can be interpreted as existing code which was modified/evolved in one or more of the forks, perhaps inconsistently. The code needs not be injected into all of the forks to simulate *DA3*, as the forks missing the code can be interpreted as having lost this shared code due to *DA2*.

DA4 Code injected into the forks, with variation in injection location, can be interpreted as existing code being moved. When the code is not injected into all the forks, the forks missing this existing code can be interpreted as instances of *DA2*.

DA5 Code injected into multiple forks can be interpreted as code implemented in one fork and copied into others. Non-uniform injection, source mutation and renaming simulate that the code may be copied into a different location than in the source, and continued development may occur independently of the source.

DA6 While the generation process creates all of the forks from the same forking point (the base system), the resulting dataset can be interpreted as originating from multiple forking points. Code shared due to injection amongst a subset of the variants can be interpreted as development before a shared forking point, which may not be shared across all the forks. This activity can also be simulated by using the forks as the base system for additional executions of ForkSim.

7.5 Discussion

Advantages The primary advantage of ForkSim is that the user can precisely control the amount and type of similarities and differences among the generated forks. This allows for well-controlled evaluation of tools which analyse forks. Moreover, as the fork generation process is known and logged, the correct and complete information on the actual code similarities and differences between the forks is available. This is not the case when real-world forks are analysed.

Disadvantages One of the limitations of ForkSim is that the generated variants may have properties that differ from real forks, particularity if aggressive injection settings are used. As injection is a random process, the code-level properties do not represent meaningful development. Also, the distribution of the similar and dissimilar code might differ from real-world forks. However, to the best of our knowledge, there are no systematic studies on the amount and distribution of code similarities and differences in real-world forks. Therefore we are not able to tune our generation algorithm and its parameters to produce very realistic forks. However, as fork analysis tools likely do not behave differently for less realistic software variants, it is unlikely that this will have a significant effect on tool evaluations using ForkSim-generated fork datasets.

Unknown Similarities The similarities and differences between the forks inherited from the subject system and intentionally created by injection are exactly known. However, there will be some additional similarities between the forks which are unknown. These include: (1) clones within the original subject system which become similar code within and between the generated forks, (2) unexpected similarity between the functions, files, and directories randomly chosen from the source repository, and (3) unexpected similarity between these chosen source artifacts and the subject system.

Since these similarities are unknown, they are not included in the measurement of a tool's recall for the dataset. However, this is not a disadvantage as we are not interested in evaluating the tools for intra-project similarities. The known similarities are sufficient for measuring cross-project similarities. These similarities, however, must be considered in the measure of a tool's precision.

7.6 Use Cases

Cross-Project Similarity Tool Performance Evaluation ForkSim datasets can be used to measure the recall and precision of tools which detect similarity between software projects. It is especially attuned for tools which focus on similarity detection between software variants (e.g. forks). ForkSim datasets are ideal for this usage scenario as similarities and differences between the generated forks are known. Recall can then be measured automatically, and precision semi-automatically. Recall is evaluated by measuring the ratio of the similar code between the forks, and their relationships, the tool is able to detect and report. The recall measure considers both the similarities created by injection, and the similarities between each of the forks due to the duplication of the base system during the generation process.

How tools report similar code is likely to differ. Therefore, to evaluate recall the dataset's generation log must be mined and converted into the detection tool's output format. This process creates the tool's gold standard, i.e., its ideal and perfect output for the dataset. Recall is then the ratio of the gold standard the tool was able to produce. By building the gold standard procedurally, recall evaluation becomes automatic. The conversion procedure needs only to be written once, and reused for various datasets.

Precision can be evaluated semi-automatically. Any detected similar code which is in the gold standard can be automatically labeled as true positive. Any reported similarities which match known differences in the

dataset can be labeled as false positives. The remaining output requires manual validation to complete the measure of precision. It is sufficient to validate a random subset (large enough to be statistically significant) of the remaining output and to estimate precision from these results.

Tool Usability Study ForkSim-generated datasets are valuable for performing controlled experiments involving tools which analyse and/or visualize similarities and differences between software variants. The goal of such an experiment can be to measure the level of support for software similarity comprehension the tools provide to their users. The experiment would have the following procedure: first, a dataset of forks with known similarities is generated using ForkSim. Then, the study participants, divided into a few groups, use the tools to analyse the dataset and report their findings. Each user group uses a different tool to solve the same tasks. For example, the participants can be asked a set of questions related to the similarity of the analysed variants, which they should answer by discovering and understanding the similarities using the given tool. The tasks should be designed to evaluate a specific aspect of the tools, e.g. their usability or the appropriateness of the used similarity visualizations. By checking the correctness of the answers and recording the amount of needed effort and/or time, user group performance is quantitatively measured. In this way, the effect of using the different tools is quantified, and the tools can be compared regarding the properties targeted by the tasks, such as tool usability. Alternatively, the user groups might use the same tool to solve different tasks, in order to determine which task type is easy or difficult for the users when they use the specified tool.

Adaptations For the purpose of clone management, detecting and studying clone genealogies is another important research topic, and there have been a few genealogy detectors (e.g., [113]). The technology used in ForkSim can easily be adapted to generate software versions rather than software variants for evaluating genealogy detector performance. Such an adaptation could also be used to evaluate clone ranking algorithms [142], which use multiple versions of a software system to produce a clone ranking.

7.7 Evaluation

As a demonstration of ForkSim's primary use case, tool evaluation, we evaluated NiCad's performance for similarity detection between software variants. While NiCad is a clone detector designed for single systems, it can be used to detect similarity between forks by executing it for the entire dataset and trimming the intra-project clone results from its output. To evaluate NiCad, we generated a ForkSim dataset of 5 Java forks. We used JHotDraw54b1 as the subject system and Java6 as the source repository. The generation parameters used are listed in Table 7.4. The NiCad clone detector is capable of detecting function and block granularity near-miss clones. It uses TXL to parse source elements of these granularities from an input system, and uses a diff-like algorithm to detect clones after these source elements have been normalized to remove irrelevant differences (e.g., formatting, comments, whitespace, identifier names, and more). For use

in this experiment, we extended NiCad to support the detection of clones at the file granularity.

Using NiCad, we detected the file and function clones in the dataset. NiCad was set to detect clones 3-5000 lines long, with at most 30% difference. It was configured to pretty print the source, blind rename the identifiers, and normalize the literal values in the dataset before detection. The clones were collected both in clone pair (pairs of similar files or functions) and clone class (set of similar files or functions) format. Overall, NiCad found 363 file clone classes (16,553 pairs) and 1831 function clone classes (2,198,636 pairs).

To evaluate NiCad's recall, we converted the known similarities between the forks into file and function clone classes. Each file injected into multiple forks was converted into a file clone class, as were the files contained in directories injected into multiple forks. File clone classes were created for each of the files the forks inherited from the subject system, with the files modified due to function injection trimmed from these classes. Each function injected into multiple forks was converted into a function clone class. Lastly, a function clone class was created for each function the forks inherited from the subject system. These clone classes were also converted to clone pair format.

NiCad's recall performance is summarized in Table 7.5. Recall was measured per clone granularity (file or function), and per origin of similarity (file/directory/function injection or original subject system files). As can be seen, NiCad had 100% recall for all sources of file clone classes, and 98-99% for function clone classes. If we consider clone pairs instead of clone classes, we see that the function clone detection is marginally better (+0.1%). These are very promising results for NiCad as a fork similarity analysis tool. These results are specific to the dataset's generation parameters. In future we plan to evaluate NiCad's recall performance for many datasets with varied parameters; for example, with larger and smaller max mutation values.

Due to time constraints, we did not perform a full precision analysis for this experiment. However, NiCad is known to have high precision [110]. Using known similarities, we were able to validate 20.7% of NiCad's reported file clone pairs, but only 1.46% of its reported function clone pairs. NiCad is reporting a large amount of cloned code beyond that of the known similarities. Part of this is due to unknown similarities arising from clones within the original subject system. However, a large fraction of this is due to the NiCad

Table 7.4: ForkSim Generation Parameters: NiCad Case Study

Parameter	**Value**
Subject System	JHotDraw54b1
Source Repository	Java6
Language	Java
# Forks	5
# Files	100
# Directories	25
# Functions	100
Function Size	20-100 lines
Max Injections	5
Uniform Injection Rate	50%
Mutation Rate	files: 50%, directories(files): 50%, functions: 50%
Rename Rate	files: 50%, directories: 50%
Max Mutations	15% of size in lines

Table 7.5: NiCad Case Study Recall Results

Type	File Injections	Directory Injections	Function Injections	Original Files
File Clone Class	100% (41/41)	100% (117/117)	-	100% (260/260)
Function Clone Class	-	-	98.7% (75/76)	99.4% (2869/2886)
Function Clone Pair	-	-	98.8% (332/336)	99.5% (28708/28860)

clone size settings used. A minimum clone size of 3 lines was required to ensure that all cloned functions were detected. However, small standard functions such as getters and setters are very similar after normalization, which was a source of a large number of these clone pairs. Likewise, interfaces and simple classes are likely to be detected as similar after identifier normalization. For practical usage, these small similarities would likely be filtered out in preference of the larger similarities. In summary, NiCad has very good detection performance of similarities between forks, but the quantity of output would make its usage difficult. A post-processing step needs to be added to extract the most useful and important similarity features from its output.

7.8 Conclusion

In this chapter we have introduced ForkSim, a tool for generating customizable datasets of synthetic forks with known similarities and differences. These datasets can be used in any research on the detection, visualization, and comprehension of code similarity amongst software variants. ForkSim datasets allow similarity detection tools to be evaluated in terms of recall (automatically) and precision (semi-automatically), and can be useful in experiments aiming at evaluating the usability and visualization of similarity tools. We demonstrated ForkSim using a case study evaluating NiCad's cross-project similarity detection for a set of five ForkSim-generated Java forks.

Part II

Real-World Large-Scale Clone Benchmarking

In this part, we present our work with real-world and large-scale inter-project clone benchmarking. In Chapter 4, we showed that the previous leading real-world clone benchmark, Bellon's Benchmark, is not appropriate for evaluating modern clone detection tools. While we had already delivered a high quality synthetic benchmark with the Mutation and Injection Framework, a new real-world benchmark was warranted and needed by the community. Additionally, no existing clone benchmark was appropriate for evaluating clone detection tools in the context of inter-project and large-scale clone detection, an emerging research topic that we explore in this thesis, so such a benchmark was needed by ourselves and the community.

For these reasons, we introduce BigCloneBench: our real-world big clone benchmark for evaluating all flavors of clone detection, including inter-project and large-scale clone detection. We built BigCloneBench by mining IJaDataset, a big inter-project source-code dataset, for clones of distinct functionalities. We designed a mining and validation procedure capable of building a large benchmark while minimizing the clone validation efforts and minimizing the subjectivity in the validation results. We built a big benchmark of eight million reference clones spanning the four clone types as well as the entire spectrum of syntactical similarity, including intra-project, inter-project and semantic clones. We describe our clone mining procedure, the contents and properties of our benchmark, and its usages in Chapter 8.

We used BigCloneBench to conduct a clone detection tool comparison study where we measured recall per clone type, including for each region of syntactical similarity. We measured and compared recall for intra-project vs inter-project clones, and evaluated how well the tools capture the reference clones using multiple clone-matching algorithms. We compared the results of our real-world benchmark against those from our synthetic clone benchmark to demonstrate the need for both styles of benchmarking to get a full understanding of a tool's recall performance. This study is presented in Chapter 9.

To make this benchmarking procedure accessible to the community, we distilled our tool evaluation experiment procedure into a customizable framework called BigCloneEval. This framework makes the execution of recall evaluation experiments with BigCloneBench easy, and handles tool execution, tool scalability, and recall measurement automatically for the user. The evaluation experiments are customizable, including a plug-in architecture for using custom clone-matching algorithms. Importantly, BigCloneEval creates a reference standard for tool evaluation with BigCloneBench. BigCloneEval is presented in Chapter 10.

Later in this thesis (Part III, Chapter 12), we use BigCloneBench to evaluate our CloneWorks clone detector for large-scale clone detection, including the measurement of recall, precision, execution time and scalability.

Chapter 8

BigCloneBench

There are multiple flavors of clone detection tools. Classical clone detection tools locate syntactically similar code within a single software system or small repository. These tools have been traditionally used to cancel out the effects of ad-hoc code reuse (e.g., copy and paste) [108] in software systems. Semantic clone detectors locate code that implements the same or similar functionalities. These tools target the clones the classical detectors miss due to a lack of syntactical similarity. Recently, new applications for clone detection and search have emerged relying on detected clones among a large number of software systems. Since classical clone detection tools do not support the needs of such emerging applications, new large-scale clone detection and clone search algorithms are being proposed as an embedded part of the emerging applications. For example, large-scale clone detection and clone search (e.g., [81]) is used to find similar mobile applications [22], intelligently tag code snippets [97], find code examples [66], and so on.

A limitation with existing benchmarks is that they only target the classical clone detectors which focus on the detection of syntactically similar intra-project clones. They typically only consider the clones within a couple of software systems. Many "flavors" of clone detection cannot be evaluated by these benchmarks. Semantic clone detectors require a benchmark of semantically similar clones with a wide range of syntactical similarity. Large-scale clone detectors require a benchmark with many inter-project clones. Clone search algorithms must be evaluated against large clone classes. By targeting a single benchmark scope, the benchmark becomes limited to a specific sub-class of clones. The community needs a standard benchmark that covers the full range of clone types and clone detection applications.

In this chapter, we introduce BigCloneBench, a large-scale clone benchmark of true and false clones in IJaDataset 2.0 [4] (25,000 subject systems, 2.3 million files, 250MLOC). Unlike previous real-world benchmarks (namely Bellon's Benchmark [13]), we did not use clone detectors to build our benchmark. Rather, we mine IJaDataset for clones of frequently used functionalities. We used search heuristics to automatically identify code snippets in IJaDataset that might implement a target functionality. These candidate snippets are manually tagged as true or false positives of the target functionality by expert judges. The benchmark is populated with the true and false clones oracled by the tagging process. We use TXL-based [27] automatic source transformation and analysis technologies to typify these clones and measure their syntactical similarity.

BigCloneBench contains 8.9 million true clone pairs of 43 distinct functionalities, which can be used to

measure the recall of all flavors of clone detection. The benchmark contains many intra-project clones for evaluating classical clone detection tools. Every clone is of a functionality, with wide variety of syntactical similarity, which makes it ideal for evaluating semantic clone detection tools. Its large number of inter-project clones makes it an ideal target for evaluating large-scale detectors. It contains large clone classes of distinct functionalities, which can be used as targets for evaluating clone search algorithms. While not all tools scale to large-scale, they can be evaluated for BigCloneBench by executing them for subsets of the benchmark within their scalability constraint. As a standard benchmark, BigCloneBench is the ideal target for comparing the execution time and scalability of clone detection tools. BigCloneBench also documents 288 thousand known false positive clones discovered during the mining process. These can be used to evaluate the accuracy of the clone detectors, but cannot replace a traditional measurement of precision by manual validation of a clone detection tool's output. We have focused our efforts on building a benchmark for measuring recall for any flavor of clone detection.

This chapter is an updated and extended version of our manuscript [125] "Towards a Big Data Curated Benchmark of Inter-Project Code Clone" which was published in the International Conference on Software Maintenance and Evolution, ©2014 IEEE. I was the lead author on this work, and my co-authors include Judith F. Islam, Iman Keivanloo, Chanchal K. Roy and Mohammad Mamun Mia. Iman Keivanloo and Chanchal K. Roy acted as supervisors for this project, while Judith F. Islam and Mohammad Mamun Mia contributed functionality selection and clone validation efforts. The publication has been updated to reflect the latest work on BigCloneBench, and re-formatted for this thesis.

The rest of this chapter is organized as follows. In Section 8.1 we discuss the related work. In Section 8.2 we describe our methodology for building BigCloneBench, in Section 8.3 we discuss our efforts executing this procedure, and in Section 8.4 we overview the contents of the final benchmark. Then in Section 8.5 we describe how the benchmark can be used to evaluate clone detection tools, and in Section 8.6 we describe how it can be used to evaluate clone search tools. We close with a description of the distribution of the benchmark in Section 8.7, the threats to the validity of the benchmark in Section 8.8, and our conclusions in Section 8.9.

8.1 Related and Previous Work

Benchmark experiments have been performed that measure the recall and precision of classical clone detection tools that scale to a single system. However, measuring recall has traditionally been very challenging. Some experiments have ignored recall, and measured tool precision by manually validating a small sample of a tool's candidate clones [38,52,70,76,82]. Other experiments have tackled the recall problem by accepting the union of multiple tools' candidate clones as the reference set, possibly with some manual validation [13,18,35,94,112]. For some experiments, very small subject systems were manually inspected for clones [18, 71, 110]. An ideal oracle could be made if all the pairs of code fragments in a subject system were inspected. However, this

is not feasible except for toy systems. For example, when considering only clones between functions in the relatively small system Cook, there is nearly a million function pairs to manually inspect [137].

Large-scale (Big Data) analysis is a very popular and rewarding field in both industry and academia. As the benefits and utility of large-scale analysis has become clearer [87], so has the number of technologies (e.g., [6, 29, 36, 37, 132]) that enable it. To develop, improve, and compare these technologies, quality benchmarks are needed. This need has been recognized by the international community in such conferences or workshops as Big Data Benchmarking [140], which began in 2010. There has been significant efforts in evaluating large-scale analysis technologies [5, 7, 25, 40, 98, 101]. For example, BigBench [40] models a typical large-scale scenario and generates a large-scale benchmark problem. The major large-scale technologies could be compared using BigBench.

In contrast, BigCloneBench is a domain-specific large-scale benchmark for evaluating all types of clone detection and clone search technologies, especially those that scale to large-scale. It can be used to measure recall, estimate precision, and compare the execution time and scalability of clone detection and search tools. The benchmark consists of a curated collection of true and false clone pairs in the large-scale inter-project repository IJaDataset. Recall and precision are measured by comparing the tool's output against the benchmark. Execution time and scalability are compared by using IJaDataset as a common target for the detection tools.

While the Mutation and Injection Framework could be adapted to large-scale by injecting artificial clones into them, the importance of evaluating the tools with real data is widely discussed [12, 75, 77, 128]. Krutz and Le [77] oracled all method pairs between randomly selected files in a subject system using several judges. While their data has high confidence, their benchmark is very small, only 66 method clone pairs.

These existing benchmarks are not suitable for evaluating the other flavors of clone detection tools, for example, the emerging large-scale clone detection algorithms. The existing benchmarks are too small, and only consider intra-project clones from a handful of subject systems. Large-scale clones span thousands of subject systems, and inter-project clones may have significantly different properties from intra-project clones. In this paper we present a large-scale benchmark that contains millions of inter-project clones, spans thousands of subject systems, was built without the use of clone detectors, and has a very clear oracling procedure.

8.2 Methodology

Our benchmark consists of clones of specific functionalities in IJaDataset. We built this benchmark by mining IJaDataset [4] for code snippets that implement candidate functionalities, which enables us to identify true and false clone pairs of the four primary clone types in IJaDataset. For each of the functionalities considered, the five step process depicted in Figure 8.1 is executed.

In the first step, we select a functionality that is possibly used in open-source Java software, and identify

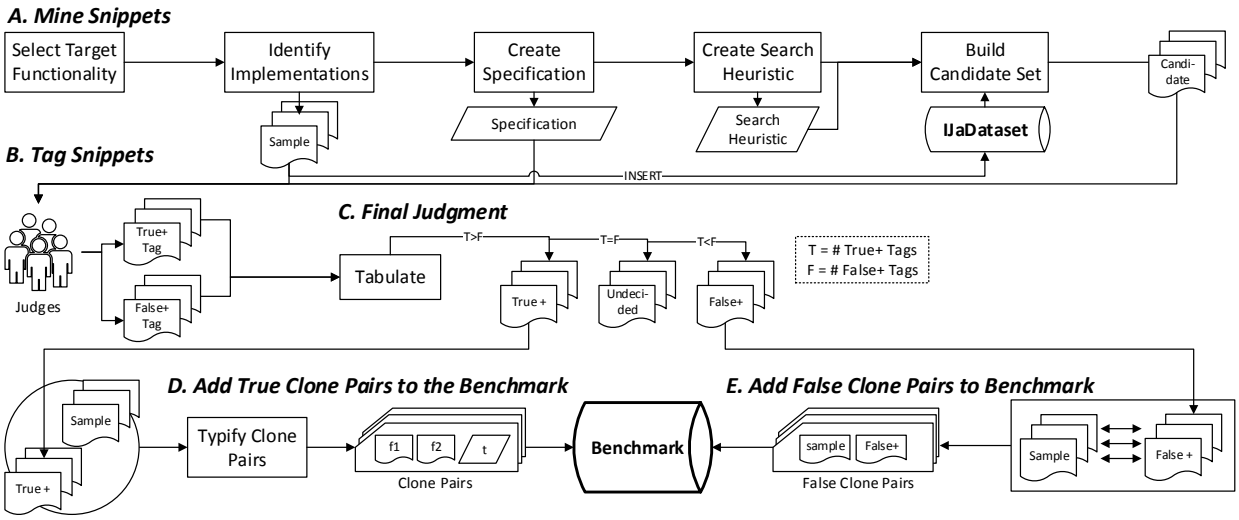


Figure 8.1: Methodology (Executed Per Functionality)

some of the possible implementations of that functionality. Using these implementations, we design a search heuristic to identify snippets in IJaDataset that *might* implement the functionality. In the second step, these candidate snippets are manually tagged by one or more expert judges as true or false positives of the target functionality. In the third step, the tags are tabulated per candidate snippet, and the snippets are judged as true or false positives considering the majority opinion of the judges. In the case of a tie (even number of judges) the candidate snippet is judged "undecided" and left unknown until additional tagging is performed. In the fourth and fifth steps, the oracled true and false positive snippets are used to populate the benchmark with the oracled true and false clone pairs. This process is repeated for a number of target functionalities, creating a live benchmark that improves with each iteration.

This methodology can be executed for any snippet granularity (method, block, etc) and programming language. We target the method granularity as it is the granularity supported by the most tools [104, 106, 108, 109], and because methods nicely encapsulate functionalities [106]. We target the Java programming language because it is the language most commonly supported by clone detection tools, and because of the availability of IJaDataset. Additionally, Java is a very popular language discussed on on-line platforms such as StackOverflow and GitHub, and we expect many cloned functionalities between distinct software projects in IJaDataset.

In the following subsections we explore the individual steps of the methodology in detail. We highlight how these steps were executed for the functionality "Shuffle an Array in Place" as our running example.

8.2.1 Mining Code Snippets

In this step, we select a target functionality and mine IJaDatset for code snippets that implement the functionality. IJaDataset contains 24 million (method granularity) snippets, equating 0.288 quadrillion method pairs, which is far too many to inspect manually. In order to reduce our manual efforts, we construct a search

heuristic to identify the snippets that *might* implement the target functionality. This smaller set of candidate snippets are manually inspected in Section 8.2.2. Snippet mining occurs over five sub-steps as follows:

(1) Select Target Functionality: We begin by selecting a functionality we believe is needed in open-source Java projects as our *target functionality*. We research and select the functionality by browsing Internet resources used by open-source developers, such as: Stack Overflow, Java tutorial sites, standard library documentation, and popular 3rd party library documentation. We draw from our own development experiences, we ask developers for sample functions, and we investigate sample open-source systems using the GitHub API [102]. We select a functionality we believe will appear many times in IJaDataset.

As a running example, we chose the functionality "Shuffle an Array in Place". Developers sometimes need to randomly shuffle the contents of an array. This is reflected in the inclusion of a shuffle method for List data structures in Java's standard library. However, no such method is provided for array structures, and sometimes developers need to shuffle arrays that are too large to be stored or temporary transitioned into a List type object. Thus, while there is no guarantee, we expect IJaDataset to contain methods that implement an in-place shuffle for arrays.

(2) Identify Implementations: Before we design a search heuristic, we need to identify how the functionality might be implemented in Java. We review Internet discussion (e.g., Stack Overflow) and API documentation (e.g., JavaDoc) to identify the common implementations of the target functionality. These resources are frequently used by open source developers, so we expect similar implementations to appear in IJaDataset. Implementations may use different support methods, data structures and APIs, or they may express the algorithms using different syntactic constructs (e.g., different control flow statements).

For each of the identified implementations, we collect a sample snippet that uses the implementation to achieve the functionality. The sample snippets are minimum working examples, and include only the steps and features necessary to implement the functionality. The sample snippets are added to IJaDataset as additional crawled open-source code. These play a role in the identified true and false clone pairs. They are also used to improve the efficiency and accuracy of snippet tagging.

For the running example, our research found that the most common implementation of an in-place array shuffle is the Fisher-Yates shuffling algorithm. It shuffles an array in place by iterating through an array and randomly swapping the value at the current index with the value at a randomly selected index. If iteration begins from the start of the array, the random index is chosen as an index greater than the current. The reverse is done if iteration begins at the end of the array. Otherwise, implementations differ in the array data type they shuffle (primitive type or object type), and which syntactical loop structure they use. We chose three sample snippets that exemplify these differences. One of these snippets is shown in Figure 8.2

(3) Create Specification: We create a formal *specification* of the functionality, including the minimum steps

```
public static <T> void shuffle(T[] a) {
  int length = a.length;
  Random random = new Random();
  for(int i = 0; i < length; i++) {
    int j = i + random.nextInt(length-i);
    T tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;}}
```

Figure 8.2: Sample Snippet: Shuffle Array

or features a snippet must realize to be a true positive of the target functionality. We derive our specification from our research into the functionality, as well as the sample snippets. Our specification is what we believe the average developer would expect the functionality to minimally perform. We validate this with our taggers, who are developers and software engineering researchers. The specification is designed to be inclusive to the possible implementations of the target functionality. It should not preclude any implementation, including any that may not have been identified in step 2.

For our running example, and from our research into its implementations and sample snippets, we decided our formal specification should simply be "shuffle an array of some data type in place". This specification is open enough that it accepts any variation of Fisher-Yates, but also will accept any snippet that shuffles an array in place using an alternate algorithm.

(4) Create Search Heuristic: We create a search heuristic to locate snippets in IJaDataset that might implement the target functionality. Drawing from the identified implementations, the sample snippets, and the specification, we identify the keywords and source code patterns that are intrinsic to the implementations of the functionality. We expect snippets that implement the functionality to contain some combination of these keywords and source patterns. These keywords and source patterns are implemented as regular expression matches and are combined into a logical expression. Keywords and patterns that are expected to appear together in an implementation are 'AND'ed, while groups of keywords and patterns that are from different implementations are 'OR'ed. The heuristic returns true if an input snippet satisfies the logical expression.

We design the heuristic with respect to the sample snippets (Step 2). We try to select keywords and source patterns that not only appear in our identified implementations, but may also appear in unidentified implementations. We design the heuristic to balance two opposing constraints: (1) the heuristic should identify as many true positive snippets as possible, but (2) should not identify so many false positives that the judges are overburdened in their tagging efforts. We balance completeness and required tagging efforts, preferring completeness. We try several search heuristics and review their results before choosing the one that provides the best balance.

For our running example, we target its search heuristic for the identified Fisher-Yates implementations.

As mentioned in Step 2, the Fisher-Yates algorithm iterates through the array and swaps the current index with a randomly selected index, with the range of the selected index depending on if iteration begins at the start or the end of the array. If looping from the start, then the selected index is chosen using the source code pattern: "*j = i + random.nextInt(length-i)*". If looping from the end, the pattern is instead: "*j = random.nextInt(i+1)*". We therefore used both of these cases using an OR clause. We implemented the patterns as regular expressions, allowing any valid white space as well as any valid alternative variable names.

(5) Build Candidate Set: The search heuristic is executed for every snippet in IJaDataset to identify possible *candidate snippets* that might implement the target functionality. For the current release we search all method granularity snippets. Manual inspection is required to identify which are true or false positives of the target functionality, as discussed in Section 8.2.2 below.

For our running example, the search heuristic we designed in Step 4 identified 281 snippets that include one of the index selection source code patterns. These form our candidate set for the "Shuffle Array in Place" functionality.

8.2.2 Tagging Snippets

Judges manually tag the candidate snippets as true or false positives of the target functionality. The judges are provided the specification of the functionality and its sample snippets. They are instructed to tag any snippet that meets the specification as a true positive of the target functionality. They tag any snippet that does not fully satisfy the specification as a false positive of the target functionality. True positives may exceed the specification by performing additional related or unrelated tasks. Due to the sizes of the candidate sets, we recommended the judges spend on the order of tens of seconds to make their decision. The judges reported that they needed 10-30 seconds to tag snippets of the simpler functionalities, and 30-60 seconds to tag snippets of the more complex functionalities. We comment on the accuracy of the judges in Section 8.3.

The judges tag the snippets using an application designed to improve their tagging efficiency and accuracy. The tagging application shows the current snippet to be tagged. The snippet is pretty-printed and has syntax highlighting to improve readability. The search heuristic as well as any other relevant keyword or source code patterns are highlighted to help the tagger locate the target functionality in a true positive snippet faster. The tagger can also view the snippet in its original file. While external to the snippet, the surrounding code may help the tagger understand the snippet faster. The main window of this application is shown in Figure 8.3.

Alongside the current snippet to be tagged, the tagging application provides the specification of the target functionality and its sample snippets. By displaying the specification within the application, the tagger can quickly review and reference it, which may reduce tagging errors. By providing access to the sample snippets, the tagger remains familiar with common implementations of the functionality. This may help the tagger

recognize similar true positives faster. Of course, the taggers fully investigate all snippets, even if they do not resemble one of the sample snippets.

Once the taggers have made their judgment, they select either true or false positive. This decision is recorded to an output file, including the snippet specification (file name and line numbers), the judgment, the tagger's name, when the tag was made, and how long the tagger reviewed the snippet before making their decision. The input file contains the specification, the sample functions, the keywords/patterns to highlight, and the snippets to be tagged.

For our "Shuffle Array in Place" example, three judges tagged the candidate set. Together they required eight hours of manual effort to tag this functionality, including both tagging time and personal research time to become an expert in the functionality. This effort produced 222 true positive tags and 621 false positive tags for the 281 candidate snippets.

Our benchmark was built by nine judges. All of the judges have considerable experience in Software Engineering, and are either clone researchers or have taken a software engineering class that has discussed clones. Before starting a case, the judges research the functionality they are to tag. They familiarize themselves with how the functionality can be implemented in Java, as well as our formal specification and sample snippets. The judges become experts in that functionality so they can tag the functionality with high confidence. In Section 8.3 we use the results of their efforts to demonstrate their accuracy.

We incrementally increased our number of judges. We began with three judges who were trained on sample data. The training data was discarded and not used in the benchmark. This phase was used to formalize the tagging procedure using the taggers' experiences and feedback. They became our primary judges. We then brought in additional judges to increase data confidence. The primary judges taught the new judges the process and tools. The new judges began on the simpler functionalities to gain experience before tagging the more complex functionalities. Each case has been tagged by at least one of the primary judges.

8.2.3 Final Judgment

Final judgment for the snippets is determined by considering the tagging of all the judges. A snippet is considered a true positive of the target functionality if it received more true positive tags than false positive tags. Likewise, it is considered a false positive if it received more false positive tags than true positive tags. If the snippets are tagged by an even number of judges then some snippets may have an equal number of true and false positive tags. These are tagged as "undecided" and are not included in the benchmark. An additional judge is required to break the tie. The use of multiple judges decreases the risk and effects of errors or bias in the tagging. The confidence in the judgment of a particular snippets depends on both the number of judges who tagged the snippet and the absolute difference between the number of true positive and false positives tags the snippet received. The true and false positive judged snippets, along with the sample snippets, are used to populate the benchmark with the oracled true and false clone pairs.

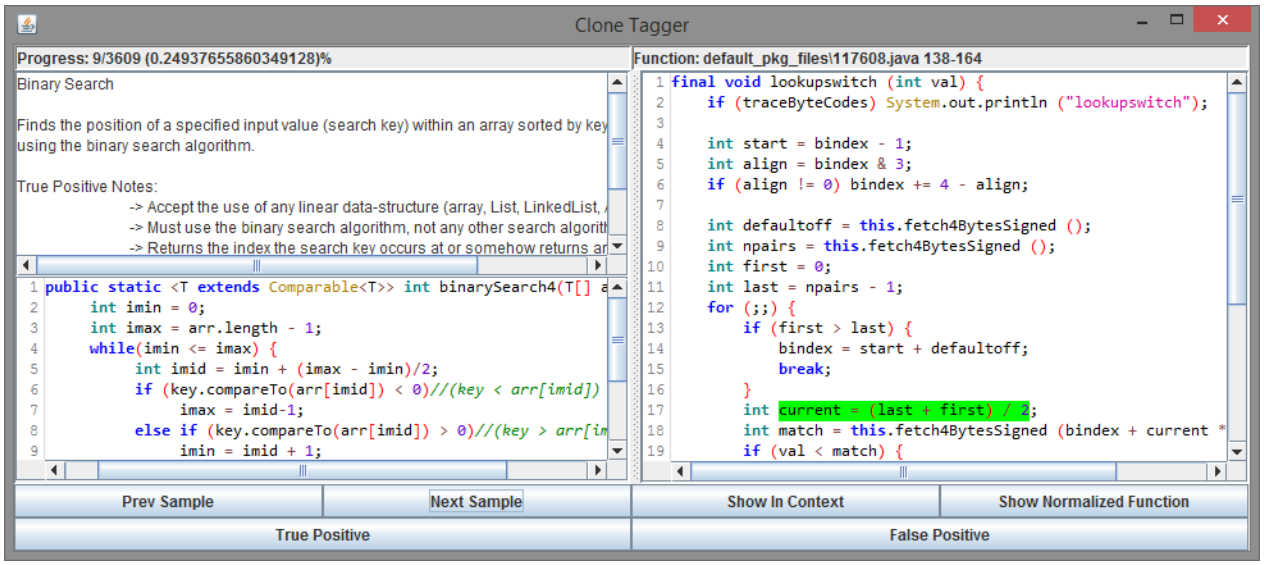


Figure 8.3: Snippet Tagging Application

For our "Shuffle Array in Place" example, we tabulated the true and false positive tags per snippet. By majority vote, 77 snippets were judged as true positives, and 204 as false positives. The three judges agreed upon 54 of the true positives and 190 of the false positives. One judge was in disagreement with the other two for 13% of the snippets, either due to subjectivity or tagging errors.

8.2.4 Adding True Clone Pairs to Benchmark

From the tagging process, we know that the true positive snippets (from the final judgment, Section 8.2.3) and sample snippets (chosen in Section 8.2.1) of a target functionality form an oracled true clone class of snippets that implement the same functionality. If we have p true positive snippets and s sample snippets that implement a target functionality, a true clone class of $s+p$ snippets, then this results in $(s+p)(s+p-1)/2$ oracled true clone pairs of that functionality in IJaDataset. Each of these clone pairs implement the target functionality, and are one of the four primary clone types. To enrich the benchmark with metadata, we typify the clones, measure their syntactical similarity, measure original and pretty-printed clone size, and measure our confidence in the clones' oracling.

For our running example, final judgment found 77 snippets that implement "Shuffle an Array in Place". With the 3 sample snippets, this is an oracled clone class containing 80 code snippets. A total of $(80)(80-1)/2 = 3,160$ oracled clone pairs implementing this functionality. These clones were added to the benchmark and enhanced with metadata as described below.

Clone Type and Syntactic Similarity. We created an automatic TXL-based [27] tool for typifying clones and measuring their syntactical similarity. A clone pair can be typified as Type-1 or Type-2 if the snippets become textually identical after the appropriate source normalization. Type-1 normalization includes removing comments and a strict pretty-printing. Type-2 normalization also includes the systematic renaming

of identifiers, and the replacing of literals with default values (e.g., numerics to 0, strings to "default", and so on). If the snippets are not identical after these normalizations then, because they implement the same functionality, they are either Type-3 or Type-4. For such cases we examine and report their syntactical similarity.

We measure the syntactical similarity of the clones using both a line-based and token-based metric after full normalization. This includes the removal of comments, a strict pretty printing, the renaming of all identifiers to a common value (e.g., 'X') and the change of all literal values to a common value (e.g., '0'). This blind normalization of identifiers and literals is needed as their values will not align due to changes at the statement level. The similarity metric measures the minimum ratio of the lines or tokens one snippet shares with another after normalization. Matching lines or tokens are identified using Unix *diff*.

These clone pairs implement the same functionality, so they are Type-3 if they are also syntactically similar, or Type-4 if they are syntactically dissimilar. However, there is no consensus on the minimum similarity of a Type-3 clone pair, so it is difficult to separate the Type-3 and Type-4 clone pairs in our case. We accept this ambiguity and instead divide the Type-3 and Type-4 clone pairs into three categories based on their syntactical similarity values: Strongly Type-3, similarity in range $[0.7, 1.0)$, Moderately Type-3, $[0.5, 0.7)$, and Weakly Type-3+4, $[0.0, 0.5)$. Where $[0.70, 1.0)$ denotes the range from and including 0.70 up to but not including 1.0. This division by similarity is useful for measuring a tool's recall for different contexts. Since we use two similarity metrics, this division can be based on either, or on the averages of the two metrics.

We define strongly Type-3 clones as those that are at least 70% similar. This is the region we expect most syntactical detectors to operate in. These clones are very similar, but contain some statement-level differences. The moderately Type-3 clones share at least half of their syntax, but contain a significant amount of statement-level differences. Syntactical clone detectors typically do not operate in this range because there is a higher chance that code snippets with only 50-70% shared syntax are only coincidentally similar. Detectors may need some level of semantic awareness to operate in this similarity region without diminished precision. We define the clones that share less than 50% of their syntax as weakly Type-3 or Type-4 clones.

Clone Size. We measure each clone pair's original and pretty-printed clone size. Clone size includes the minimum and maximum sizes of the clone pair's code snippets, measured by line and by token.

Oracling Confidence. With each clone pair we report the minimum number of judges that examined the clone's snippets, and the minimum difference between the number of true positive and false positive tags the snippets received by the judges. This allows a benchmark user to select a subset of the benchmark that meets some minimum data confidence. We use this metadata to comment on the accuracy of the judges in Section 8.3. The use of multiple judges is atypical in major clone benchmarks (e.g., [13]).

8.2.5 Adding False Clone Pairs to Benchmark

From manual inspection (Section 8.2.2) we know that the snippets judged (Section 8.2.3) as false positives of a target functionality do not implement the target functionality, while the sample snippets (Section 8.2.1) implement only the target functionality. So each pair of sample snippet and false positive snippet for a target functionality is an oracled false clone pair in our benchmark. If we have s sample snippets for a target functionality, and found f false positives snippets of this functionality, then this results in $s \times f$ false clone pairs. While the false clones might share some syntactical similarity, our manual tagging process has validated that this similarity is coincidental. As with the true clone pairs, we add clone size and oracling confidence metadata to the false clone pairs.

For our running example, final judgment found 204 snippets that do not implement "Shuffle an Array in Place", while we know the 3 sample snippets are minimum examples that only implement this functionality. This is a total of 612 false clone pairs that implement different functionalities, but could contain some coincidental syntactical similarity. These false clone pairs were added to the benchmark.

8.3 Snippet Tagging Efforts

For the creation of our benchmark, 78 thousand snippets were tagged across 43 distinct functionalities, an effort that required 514 hours of manual tagging by nine judges. The results of the snippet tagging efforts are summarized in the left sub-table of Table 8.1 (under the Snippet Tagging sub-heading) per functionality and overall. The sub-table tabulates the number of sample snippets added to IJaDataset for a functionality, and the number of snippets judged as true positive, false positive or undecided by the judges. The right side of the sub-table summarizes the efforts involved, including the number of unique judges assigned to a functionality, the total number of snippet tags they produced, and their combined tagging hours. When multiple judges were used, each judge may not have tagged every snippet. We tried to maximize the overlap to improve data confidence and to measure tagging accuracy.

Tagging hours includes time spent by the judges actively tagging the snippets, and time spent by the judges reviewing and understanding the functionality, specification and sample snippets. Active tagging time per snippet was measured by the tagging application, while the other time requirements were estimated based on feedback from the judges. Tagging hours does not include time spent choosing and researching the functionalities, designing and testing search heuristics, and preparing the data for the tagging application. We estimate that approximately two hours was spent per case, for a total of 84 hours. In total at least 600 hours was spent building this benchmark. Note that this does not include all of the research and development efforts.

In order to measure the reliability of our judges, we had multiple judges tag the same snippets. By analyzing the agreements or disagreements between judges, we can estimate the frequency in which subjectivity or human error affects the tagging data. In total, we had multiple judges tag 9,533 of the 77,933 snippets,

Table 8.1: Snippet Tagging and Benchmark Contents Summary

Functionality	Snippet Tagging							Benchmark Contents					
		Judged Snippets			Tagging Effort			True Clone Pairs $(s+p)(s+p-1)/2$					
	Snippets, s	True+, p	False+, f	Undecided	# Judges	# Tags	Hours	T1	T2	Strong T3	Moderate T3	Weak T3 & T4	False Clone Pairs, $s*f$
Download From Web	3	910	12946	0	1	13856	55.1	1554	9	1599	18359	394807	38838
Secure Hash	1	1342	4564	0	1	5906	27.3	632	587	6396	81903	811635	4564
Copy File	6	3084	34018	0	1	37102	166.6	13805	3116	9297	95191	4651096	204108
Decompress zip archive.	2	7	6	22	2	70	3.4	0	0	1	2	33	12
Connect to FTP Server	11	213	381	92	2	1372	7.3	4	0	94	542	24336	4191
Bubble Sort Array	14	158	1142	237	2	3074	15.9	40	4	466	4167	10029	15988
Setup SGV	1	23	78	0	1	101	1.5	3	7	5	11	250	78
Setup SGV Event Handler	1	10	1272	0	1	1282	7.0	0	0	0	1	54	1272
Execute update and rollback.	2	751	905	44	2	1847	12.2	152	66	1288	13470	268152	1810
Initialize Java Eclipse Project.	1	22	0	0	2	44	2.6	0	0	8	1	244	0
Get Prime Factors	2	22	76	0	3	294	5.8	5	3	27	72	169	152
Shuffle Array in Place	3	77	204	0	3	843	8.4	8	5	290	689	2168	612
Binary Search	4	438	3075	96	3	4968	23.3	273	7	1013	13845	82323	12296
Load Custom Font	3	24	7	1	4	128	4.7	0	0	4	10	337	21
Create Encryption Key Files	3	18	150	2	3	304	7.5	0	0	0	10	200	450
Play Sound	3	36	41	4	2	162	2.7	5	0	10	89	637	123
Take Screenshot to File	1	104	296	55	2	910	5.2	17	2	19	304	5118	296
Fibonacci	1	211	0	0	3	633	3.6	11194	1	1396	5406	4369	0
XMPP Send Message	1	24	40	1	2	130	2.3	1	0	2	20	277	40
Encrypt To File	1	74	106	8	2	248	3.8	1	1	6	89	2678	106
Resize Array	1	439	23	77	2	1078	53.0	200	26	757	5926	89671	23
Open URL in System Browser	1	387	45	5	2	874	3.7	40	20	584	9513	64921	45
Open File in Desktop Application	1	104	177	5	2	572	3.8	4	3	16	474	4963	177
GCD	3	20	117	13	2	278	3.9	1	0	35	61	156	351
Call Method Using Reflection	4	415	126	0	1	541	2.0	3557	8	663	496	82847	504
Parse XML to DOM	2	195	52	0	1	247	1.7	199	10	193	558	18346	104
Convert Date String Format	1	58	154	13	2	330	5.6	15	2	22	79	1593	154
Zip Files	1	1425	691	12	3	2518	10.1	209	33	1022	33943	980818	691
File Dialog	4	476	0	24	2	850	3.9	1926	107	214	3054	109659	0
Send E-Mail	1	239	33	1	2	378	3.9	453	2	416	1171	26638	33
CRC32 File Checksum	1	282	36	20	3	796	5.8	62	7	97	765	38972	36
Execute External Process	2	464	36	10	2	668	4.2	1485	25	921	2238	103676	72
Instantiate Using Reflection	1	861	44	0	1	905	3.2	266	41	468	2587	367729	44
Connect to Database	2	204	3	3	2	365	2.9	90	34	406	515	20070	6
Load File into Byte Array	1	158	202	0	1	360	2.6	42	4	95	1153	11267	202
Get MAC Address String	2	21	2	16	2	78	2.5	0	0	1	17	235	4
Delete Folder and Contents	2	274	73	4	3	681	6.0	253	27	1169	6202	30299	146
Parse CSV File	1	201	49	0	1	250	1.8	482	2	824	614	18379	49
Transpose Matrix	1	542	50	54	2	1292	5.2	484	65	3909	19011	123684	50
Extract Matches Using Regex	1	502	3	0	2	1010	5.1	47	19	109	3754	122324	3
Copy Directory	2	150	183	30	2	513	7.6	3	3	278	977	10215	366
Test Palindrome	1	167	221	0	1	388	1.7	10879	0	152	763	2234	221
Write PDF File	1	158	129	38	4	780	7.6	1	3	168	1103	11286	129
Total	101	15290	61756	887	10	89026	513.9	48392	4249	34440	329155	8498894	288367
		77933						8915130					

T3 Categories by Syntax Similarity Ranges - Strong: [0.70,1.0) **Moderate**: [0.50,0.70), **Weak:** [0.0,0.50)

a coverage of 12.2%. We distributed the snippets selected for multiple tagging across 75% of the functionalities. When two judges tagged the same snippet, they disagreed for 10.8% of the snippets. For three judges disagreement was 20.4%, and for four judges disagreement was 18.8%. While more judges increases the chance we discover subjectivity, it also increases the likelihood that one of the judges created an error. Considering all of the snippets tagged by multiple judges, regardless of the number of judges, at least one judge disagreed with the others for 14.5% of the snippets. From this, we estimate for the snippets tagged by only a single judge, 14.5% of them are affected by subjectivity or tagging errors. We believe this is an acceptable value [137]. However, since no previous major clone oracle (e.g., [13]) has significantly commented on the accuracy of their judges, we have no basis of comparison.

8.4 The Benchmark

From the tagging data, we were able to identify 8.9 million true clone pairs (Section 8.2.4) and 288 thousand false clone pairs (Section 8.2.5) across 43 functionalities in IJaDataset. The true and false clone pair contents of BigCloneBench are summarized in the right sub-table of Table 8.1 (under the Benchmark Contents subheading). Each of the true clone pairs listed is a clone of the listed functionality. The tagging efforts locate $\frac{1}{2}(s+p)(s+p-1)$ oracled clone pairs per functionality, as described in Section 8.2.4, where s is the number of samples, and p is the number of snippets judged as true positive. We summarize the true clones per clone type, with Type-3 and Type-4 clones divided by their syntactical similarity as described in Section 8.2.4. For this sub-table we averaged the line and token-based metrics to create the divisions. Each of the false clone pairs listed is a pair of code snippets that do not share functionality. The tagging efforts locate $s * f$ oracled false clone pairs as described in Section 8.2.5, where f is the number of snippets judged as false positive.

Clone size can be measured as the size of the smaller snippet, the size of the larger snippet, or the average sizes of the snippets. Across the benchmark, the average clone pair is 18 lines (by the smaller snippet), 51 lines (by the larger snippet), and 34 lines (average size of the snippets). Clone size is important as minimum snippet size is a common parameter of clone detection and search algorithms. We have found that tools typically have a default minimum snippet size of 6-15 lines. In our benchmark, 96% of the clones have snippets no smaller than 6 lines, 79% have snippets no smaller than 10 lines, and 53% have snippets no smaller than 15 lines. Tools can be comfortably configured for a minimum clone size of 6 lines for evaluation against the entire benchmark. Alternatively, the tool can be evaluated for a subset of the benchmark within its clone size range. The benchmark is large enough that such a subset will remain large for any reasonable minimum clone size.

Previous benchmarks [13,77] contain only intra-project clones, and most were built using the subject clone detectors themselves [13]. They do not measure the detection capabilities needed from clone detectors that target large-scale and/or inter-project detection. They are also missing the low similarity clones that classical syntactical clone detectors are missing. Our benchmark is needed to measure large-scale inter-project clone

detection tools accurately. It also includes clones that the classical syntactical clone detectors are missing.

8.5 Evaluating Clone Detectors

Our benchmark can be used to measure the recall of clone detection tools and estimate their precision. Recall and precision are shown in (8.1), where B_{tc} is the set of all true clone pairs in the benchmark, B_{fc} is the set of all false clone pairs in the benchmark, and D is the set of candidate clone pairs reported by the detector. Also interesting is measuring a tool's recall for subsets of B_{tc}. For example, all clone pairs of a particular functionality, all clone pairs of a particular type, all Type-3 clone pairs within a particular range of syntactical similarity, and so on. Precision is estimated as the ratio of the known clone pairs (true and false) found by the detector that are true clones. It ignores the detected clones that are unknown to the benchmark. However, the primary purpose of our benchmark is to measure recall, which has been an open problem in the community for the last decade. While the estimate of precision provides some insight, it does not replace a true measurement of precision: the manual validation of the output of a tool. However, without benchmarks like ours, it is not possible for tool developers to measure recall because they do not know which clone pairs exist in a system or repository.

$$recall = \frac{|D \cap B_{tc}|}{|B_{tc}|} \quad precision = \frac{|D \cap B_{tc}|}{|D \cap (B_{tc} \cup B_{fc})|} \tag{8.1}$$

Our benchmark is outside the scalability constraints of classical clone detection tools, which are not designed for large-scale. While these tools cannot be executed for IJaDataset in its entirety, they can be executed for subsets of the benchmark. The subsets would need to be small enough such that the tool could be executed for the relevant source files without scalability issues. The subsets could be randomly chosen, could be all the true and false clone pairs found for a functionality, or could even be the intra-project clone pairs found in one of the 25,000 original subject systems crawled for IJaDataset. High confidence could be

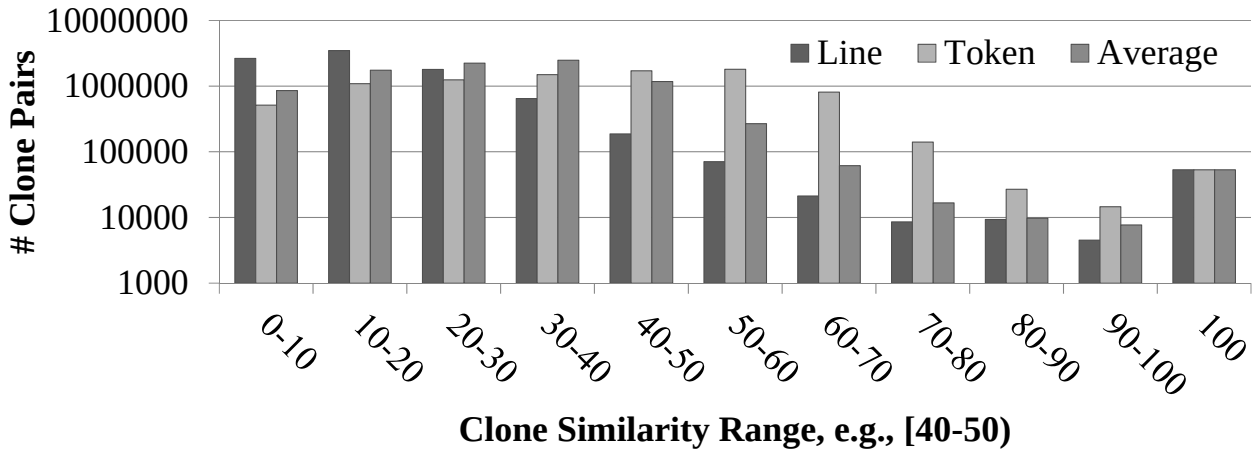


Figure 8.4: Clone Similarity Distribution

achieved by evaluating the tool for a large number of subsets.

An advantage of using our large-scale benchmark to evaluate these classical tools is clone variety. In addition to their inherit weaknesses [9, 128], classical benchmarks only consider 1-10 subject systems, which provides a limited variety of clones, especially Type-1 and Type-2. In our experience with the subject systems of Bellon's benchmark [13], the clone pairs from a single subject system are often dominated by a few large clone classes, and therefore have very little variety. In contrast, our benchmark considers 43 functionalities across 25,000 subject systems with a total of 8.9 million clone pairs. Also, our benchmark was built independently of clone detection tools.

Since our benchmark consists of clones of particular functionalities, it is very useful for evaluating semantic clone detectors (e.g., [38]). To our knowledge, there is also no significant benchmark for semantic clone detectors. While semantic clone detectors may not be scalable to large-scale, they could be executed for subsets of the benchmark. Good subsets would be the individual functionalities, or a random selection of true and false clone pairs from each of the functionalities.

While our focus was on measuring recall and precision, the benchmark can also be used as a common target for measuring clone detection execution time and scalability. Big data clone detection and search tools can be compared by their execution time for IJaDataset. Classical tools can be compared by the benchmark subset size they can handle, and their execution time for common subsets. Additionally, some large-scale clone detection tools [141] use common large-scale analysis frameworks such as Hadoop [36]. Our benchmark can be used to evaluate the execution performance (time and scalability) of these frameworks when used in a clone detection context.

8.5.1 Example Tool Evaluation: D-NiCad

While a tool evaluation experiment is out of the scope of this work, we provide a small demonstration of an example use of our benchmark. We used our benchmark to evaluate D-NiCad, a distributed version of NiCad that scales to large-scale. It uses the distributed and deterministic scalability heuristic introduced by D-CCFinder [84]. This heuristic executes NiCad for subsets of IJaDataset within its scalability limits. Across a large number of executions, NiCad is exposed to every file pair (and thus every clone) in the dataset. The executions are distributed over a number of computers. For this case study, we executed D-NiCad for a subset of IJaDataset that includes the files containing the sample snippets and tagged snippets of the first ten functionalities in Table 8.1. D-NiCad was configured to detect function clones of size 6 lines or greater for a 70% similarity threshold and full Type-1/2 source normalization.

D-NiCad's recall results are as follows: Type-1 (99.7%), Type-2 (99.6%), Strongly Type-3 (93.0%), Moderately Type-3 (0.5%), and Weakly Type-3+4 (0%). Since NiCad is a line-based tool, we separated these type 3 true clone pairs using the line-based metric. Our benchmark estimates D-NiCad's precision as 99%. Our earlier studies [128] have shown NiCad to have very high intra-project recall (99-100)% for the first three clone types. We see a marked decrease in Type-3 recall (-6%) for strongly Type-3 inter-project clones.

Our benchmark reveals that there is many true clone pairs D-NiCad misses because their similarity is below NiCad's recommended similarity threshold. This detection information can be used to improve NiCad's detection performance for large-scale inter-project clone detection. D-NiCad has strong recall and precision for Type-1, Type-2, and Strongly Type-3 clones. Ideally, future development will lower its recommended similarity threshold into the Moderately Type-3 clone range while maintaining its superb precision [110].

These results demonstrate the need for our large-scale benchmark. Classical intra-project benchmarks did not reveal these gaps in NiCad's detection [128]. Perhaps because these clones have properties that are specific to inter-project cloning, or perhaps there are edge-case gaps in NiCad's detection abilities that are not revealed by the limited number and variety of intra-project clones in a handful of subject systems [13]. A standard clone detector agnostic large-scale benchmark is needed to properly evaluate the clone detection techniques.

8.6 Evaluating Clone Search

Our benchmark can also measure the recall and precision of large-scale clone search algorithms. Given a sample snippet of one of our tagged functionalities, the clone search algorithm should return all of the snippets judged as true positives of the functionality, and none of the snippets judged as false positives. The clone search algorithm should be executed for each functionality in the benchmark, using each of a functionality's sample snippets as a target. This is one execution of the clone search tool per unique sample snippet. Recall and precision can then be measured for each functionality and sample snippet pair. The tool's general recall and precision can be measured by averaging across these cases.

For a single case, recall is measured and precision estimated as in (8.2), where f is the functionality, s is the sample snippet of f used as the search target, D is the set of detected snippets, T_f is the set of snippets judged as true positives of f, and F_f is the set of snippets judged as false positives of f.

$$recall(f,s) = \frac{|D \cap T_f|}{|T_f|} \quad precision(f,s) = \frac{|D \cap T_f|}{|D \cap (T_f \cup F_f)|} \tag{8.2}$$

8.7 Distribution

BigCloneBench and IJaDataset can be obtained from our GitHub page[1]. The benchmark is provided in database format using the schema shown in Figure 8.5. The database tables are richly populated with all the data we used, mined, created, etc. Full schema definition is available on the distribution site. The database gives users full access to our data and query power over it. However, the database is very large and may be cumbersome for simple benchmark experiments. We therefore also provide the true and false clone pairs in a simple file format. This includes the full benchmark, as well as strategic subsets of the benchmark. For

[1] `github.com/clonebench/BigCloneBench`

example, all clones of a functionality, all clones for a clone type, etc. Users with complex experiments can use the query power of the database version.

Benchmarks such as BigCloneBench are very valuable to the community. We are therefore committed to open distribution of the benchmark. In addition to full distribution of our data, we also provide access to our snippet tagging and data processing tools. We provide these so that the community may fully review our process, extend our work, or even apply similar methodologies for benchmarking similar problems.

8.8 Threats to Validity

8.8.1 Limitations in the Universality of the Mining Procedure

A contribution of this study is a mining procedure for efficiently building large-scale clone benchmarks. We specifically implemented and executed this procedure for mining Java clones at the function granularity. However, the core procedure should be applicable to other programming languages as well. In particular, we believe the procedure should work without modification for most procedural languages, such as C, C# and C++. The mining procedure could possibly be executed for other kinds of programming languages (e.g., functional, logic, etc.), but since we have not explored this, we cannot say what additional challenges might need to be overcome. Since the mining approach is based on shared functionalities, it may be more difficult to execute it for other syntax granularities, such as code blocks. Functions are ideal as they are whole units that intend to encapsulate a whole functionality.

8.8.2 Limitations in Human Judgment

Like any other manual validation task, our snippet tagging is affected by human subjectivity and error. Previous studies have shown disagreement amongst judges during clone validation [9,19,21,137]. We overcame this problem with the Mutation and Injection Framework (Chapter 3) by synthesizing clones without requiring human judgment. However, clone synthesis cannot reliably produce complex clones without the risk of the clones diverging too far from those produced by real developers. So real-world benchmarks produced in part

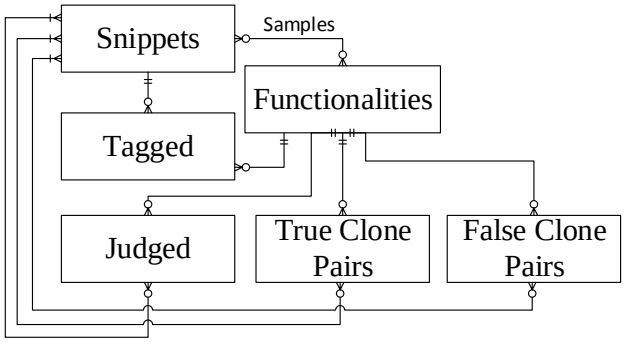


Figure 8.5: Benchmark Database Schema

by manual clone validation are required.

We attempt to reduce subjectivity and error in the creation of BigCloneBench by validating the clones indirectly. Instead of asking the judges to validate if two code fragments are similar, we ask them to validate if an individual function implements a target functionality. We then identify clones as code fragments that share functionality, and then use automatic analysis to identify the types of these clones. To reduce subjectivity in this judgment, we give the judges a clear specification on the requirements of a true positive snippet for each target functionality, and this specification is kept visible and accessible to the judges at all times in the tagging application. The judges are also trained on sample implementations of the functionality before tagging. While Charpentier [19,21] found disagreement amongst their judges in clone validation, their judges validated the clones directly, and they were not provided specific guidelines on how to make their decisions. While individual subjectivity on what constitutes a real clone is an open and unavoidable problem in clone research [109], we mitigate this by providing a clear specification of what is a true clone in BigCloneBench. Specifically, a pair of code fragments that implement the same target functionality, where that functionality is clearly defined in the benchmark.

Of course, our manual efforts are still subject to errors, and we don't claim to totally overcome subjectivity. Of the snippets tagged by multiple judges, for only 14.5% did at least one judge disagree with the others. We therefore estimate that 14.5% of the snippets tagged by only a single judge are affected by subjectivity or tagging errors. Given the difficulty of creating clone benchmark data [137], we believe this error rate is acceptable. Especially considering previous major benchmarks, including Bellon's Benchmark [13], have used only a single judge and did not comment on their accuracy. While Kurtz and Le [77] used multiple judges, their benchmark is very small, only 66 clone pairs.

8.8.3 Limitations in Clone Definitions

A limitation in all clone benchmarks is disagreement between experts on what constitutes a true code clone. This is reflected in the open-ended definition of a clone as any pair of code fragments that are similar, for some definition of similarity [11, 13]. Therefore, there is a real threat that experts may disagree on some of the reference clones included in BigCloneBench. Determining if a pair of code fragments is a true clone is known to be subjective [20, 21], and depends on the intended use-cases for the clones.

Our design of BigCloneBench aims to mitigate this threat. We provide a clear definition of the clones included in BigCloneBench: pairs of (function) code fragments that both implement some known functionality. The functionality may be a small or large part of the code fragments, and the code fragments may or may not be syntactically similar. Depending on the intended use-cases of a subject clone detector, we may not want it to detect all of the clones in BigCloneBench. For example, we may not want a tool designed to find simple refactoring opportunities to detect the clones with low syntactical similarity, as they may not be useful to this use-case. On the other hand, we would want a semantic clone detector to find these clones. The reference clones in BigCloneBench have been augmented with metadata so that the benchmark user can

select the clones relevant to the use-case they want to evaluate their subject clone detectors against.

8.9 Conclusion

In this chapter, we presented BigCloneBench, a curated benchmark of inter-project clones in IJaDataset, a large-scale source code repository. This benchmark was created using a novel functionality-based and heuristic-search clone mining approach. Unique to our benchmark is the identification of both semantically and syntactically similar clones. Unlike previous clone mining efforts (e.g., [13]), our benchmark was built independently of clone detection tools, which is the recommended approach for evaluating modern clone detection tools [9, 12, 128]. This means it is not biased or limited to the clones that detectors are able to locate. This makes the benchmark ideal for identifying weaknesses in the current detection techniques.

This release of the benchmark contains 8.9 million true clone pairs and 288 thousand false clone pairs over 43 functionalities. We show how the benchmark may be used to measure recall and estimate precision for intra-project, inter-project, semantic, and large-scale clone detection and search. With this benchmark, we have also introduced a large-scale benchmark methodology that may be useful to other domains, maybe requiring only minor adaptations to our procedure and tools.

Chapter 9

Evaluating Clone Detection Tools with BigCloneBench

In this chapter, we use BigCloneBench to evaluate the recall of ten clone detection. We measure recall per clone type, including across the entire range of clone syntactical similarity. We evaluate the tools for both single-system and cross-project detection scenarios. Using multiple clone-matching metrics, we evaluate the quality of the tools' reporting of the benchmark clones with respect to refactoring and automatic clone analysis use-cases. We compare these real-world benchmarking results against results from our synthetic benchmark, the Mutation and Injection Framework, to reveal deeper understanding of the tools and to demonstrate the need for both real-world and synthetic benchmarks. We found that the tools have strong recall for Type-1 and Type-2 clones, as well as Type-3 clones with high syntactical similarity. The tools have weaker detection of clones with lower syntactical similarity.

In summary, we address the following research questions:

RQ1 What is the recall of these tools as measured by the real-world benchmark BigCloneBench?

RQ2 How does real-world benchmarking compare to synthetic? What do the similarities and differences tell us about tool performance and benchmark accuracy?

RQ3 How does intra and inter-project clone recall differ, in particular for the case of an ultra-large dataset?

RQ4 What is the clone capture quality of the tools for refactoring and clone analysis use-cases?

This chapter is based upon our manuscript "Evaluating Clone Detection Tools with BigCloneBench" published by myself and Chanchal K. Roy and in the Research Track of the International Conference on Software Maintenance and Evolution, ©2015 IEEE. I was the lead author of this paper, under the supervision of my supervisor Chanchal K. Roy. The publication has been re-formatted for this thesis, with modifications to better fit the thesis.

This chapter is organized as follows. We overview our benchmarking procedure presented in Section 9.1. We discuss the benchmark results in Section 9.2, including comparison of the benchmarks to demonstrate the need for both real-world and synthetic clone benchmarks. In Section 9.3 we use BigCloneBench to study the recall of the tools specifically for inter-project and intra-project clone detection, and in Section 9.4 we

examine how well precisely they capture the reference clones. We close this study with a discussion of the treats to validity in Section 9.5, and our conclusions in Section 9.6.

9.1 Experiment

9.1.1 Big Clone Bench.

The contents of BigCloneBench considered for this experiment are summarized in Table 9.1. We use only the clones that are at least 6 lines and 50 tokens in length. This allows us to configure the tools appropriately for clone size. This is a typical minimum clone size used by tools [109] and previous benchmark experiments [13]. We removed the source files that are 2000 lines in length or longer and their clones. These files make up an insignificant portion of IJaDataset (6238 files), but significantly impact the execution requirements (time, memory) of the clone detectors.

Since there is no consensus on the minimum syntactical similarity of a Type-3 clone, it is difficult to separate Type-3 and Type-4 clone pairs that implement the same functionality, as in our BigCloneBench. Instead, we divide the Type-3 and Type-4 clones into four categories based on their syntactical similarity. We define **Very-Strongly Type-3** clones (**VST3**) as those with a similarity in range 90% (inclusive) to 100% (exclusive), **Strongly Type-3** (**ST3**): 70-90%, **Moderately Type-3** (**MT3**): 50-70%, and **Weakly Type-3/Type-4** (**WT3/4**): 0-50% [125]. We measure similarity as the minimum ratio of lines or tokens a code fragment shares with another after Type-1 and Type-2 normalization. Shared lines or tokens are identified by diff [34]. Most tools measure similarity by line or by token. We classify the clones into these categories using the smaller of their line and token-based clone similarity measures.

We executed the tools for IJaDataset 2.0, and measured their recall for the clones in BigCloneBench. These subject tools were generally designed for clone detection within a single software system, or a small collection of software systems. None of these tools can scale to IJaDataset on ordinary hardware. Our goal is to measure recall using a large number of clones, not to evaluate the scalability of the tools. We avoid the scalability issue by executing the tools for smaller subsets of IJaDataset that expose the tools to every clone in BigCloneBench. We executed the tools for one subset per functionality in BigCloneBench. Each subset includes every file that contains a function judged as a true or false positive of the subset's functionality during the mining process. Therefore, each subset contains a mix of true and false clones. If a subset was too large for a tool to evaluate within memory constraints (12GB), we partitioned the subset into smaller sets and executed the tool for each pair of partitions. A tool's clone detection reports were merged before

Table 9.1: BigCloneBench Clone Summary

Clone Type	T1	T2	VST3	ST3	MT3	WT3/T4
Number of Clone Pairs	35787	4573	4156	14997	79756	7729291
				7868560		

evaluation.

With BigCloneBench, we use a coverage-based clone matching metric to determine if a reference clone in the benchmark is successfully detected by a candidate clone reported by a tool. The **coverage-match**, or *c-match* for short, is based on our *covers* metric. A code fragment f_1 covers code fragment f_2 if it intersects a ratio t of the source lines of f_2, as shown in (9.1), given that the code fragments are in the same source file. A candidate clone, C, matches a reference clone, R, by the *c-match* if its code fragments cover a ratio t of the reference clone's code fragments, as shown in (9.2). When the metric is evaluated, both orderings of the candidate clone's code fragments are tested. We configured the metric with a 70% minimum coverage threshold. This is a conservative threshold, neither too strict nor too generous, that has been used in previous benchmarking experiments [13, 128]. A tool's recall is therefore the ratio of the reference clones in the benchmark that are matched by candidate clones reported by the tool, as judged by the *c-match* clone-matching metric.

$$covers(f_1, f_2, t) = \frac{min(f_1.e, f_2.e) - max(f_1.s, f_2.s) + 1}{f_2.e - f_2.s + 1} \geq t \tag{9.1}$$

$$c\text{–}match(C, R, t) = covers(C.f_1, R.f_1, t) \wedge covers(C.f_2, R.f_2, t) \tag{9.2}$$

9.1.2 Mutation and Injection Framework.

We set the framework to randomly extract 250 functions from a source repository and, from each, create 15 mutant functions using the 15 mutation operators (3,750 clone pairs). Each clone was randomly injected into 10 unique copies of a subject system (37,500 mutant systems). We used IPScanner as our subject system, and JDK6 and Apache Commons as our source repository. We constrained the benchmark to the following clone properties: (1) 15-200 lines in length, (2) 100-2000 tokens in length, and (3) mutations do not occur within the first and last 15% of a code fragment by line. The 15% mutation containment ensures that the introduced edits occur within the clone, and not on its edges. Clone detection time often scales with minimum clone size, so we used a larger minimum clone size to make execution of the tools for 37,500 systems practical. We measured the syntactical similarity of the Type-3 clones and found that they correspond to the Very-Strongly Type-3 similarity region. These are different clones than we used in our comparison with Bellon's Benchmark [128]. Here we generated function clones for best comparison with BigCloneBench, whereas we previously generated block clones for best comparison with Bellon's Benchmark.

Recall is measured by a subsume-based clone-matching metric that is parameterized with the mutation containment. For a mutation containment of 15%, the metric considers a candidate clone to subsume a reference clone even if the candidate misses the first and/or last 15% of the reference clone's code fragments. This is essentially the *c-match* metric with the added restriction that the candidate must cover the inner 70% of the reference. This restriction ensures that any candidate accepted as a match of a reference clone has captured the clone-type specific edits added to the reference by a mutation operator. Therefore, recall

measured by the Mutation Framework reflects a tool's ability to handle the specific clone edit types from the editing taxonomy.

9.1.3 Tool Configuration.

The subject tools, the clone types they can detect, and their configurations for the benchmarks, are summarized in Table 9.2. We wanted the recall measurements to reflect what an experienced user can expect with their own systems. An experienced user has explored a tool's parameters and documentation, and modifies the default settings for their use-case. We configured the tools from a user-perspective by considering: (1) the default settings, (2) the documentation, and (3) the known properties of the target benchmark, which include clone types, syntactical similarity, and clone size. We also consulted the tool developers, where available. We enable any supported Type-1 and Type-2 normalizations. We configured the tools with Type-3 sensitivity thresholds based on their defaults and documentation. While greatly lowering their syntactic similarity threshold may enable them to detect more clones in BigCloneBench [65], which contains clones across the entire spectrum of syntactical similarity, their lack of semantic awareness would also cause them to detect a large number of false positives. Users are most likely to follow the recommended thresholds, so our results reflect standard usage of the tools. If a setting was not well documented, we experimented with it to observe its effect. We avoided over-configuring or over-optimizing the tools for the benchmark, as a user would not be able to do this for their own systems. We also avoided configuring the tools in a way that would increase recall at the expense of precision. The per benchmark configurations are mostly the same, except for differences in minimum clone size. Both benchmarks have a strict minimum clone size, so the tools could be configured for clone size with confidence. While different configurations may improve recall, these configurations reflect usage by an experienced user.

9.2 Benchmark Results

In this section, we measure the recall of the tools using BigCloneBench (**RQ1**) and the Mutation Framework. We compare these results, and interpret what the similarities and differences between the benchmarks tell us about the tool performance and benchmark accuracy (**RQ2**). The recall measurements by the benchmarks, and their differences, are show in Table 9.3. Due to space considerations, we do not show Mutation Framework recall per mutation operator. Instead, we summarize recall per clone type by averaging across the mutation operators that produce a particular clone type. The Mutation Framework's Type-3 clones best match the syntactical similarity of the Very-Strongly Type-3 clones in BigCloneBench, so we compare using this Type-3 category.

Table 9.2: Subject Tools and Configurations

Tool	Types	BigCloneBench	Mutation Framework
CCFinderX [58]	1,2	Min length 50 tokens, min token types 12.	Min length 50 tokens, min token types 12.
ConQat [57]	1,2,3	Min length 6 lines, max errors 5, gap ratio 30%.	Min length 15 lines, max errors 3, gap ratio 30%.
CPD [99]	1,2	Min length 50 tokens, ignore annotations/identifiers/literals, skip parser errors.	Min length 100 tokens, ignore annotations/identifiers/literals, skip parser errors.
CtCompare [133]	1,2	Min length 50 tokens, max 6 isomorphic relations.	Min length 100 tokens, max 3 isomorphic relations.
Deckard [53]	1,2,3	Min length 50 tokens, 85% similarity, 2 token stride.	Min length 100 tokens, 85% similarity, 4 token stride.
Duplo [32]	1	Min length 6 lines. Min 1 character per line.	Min length 15 lines. Min 1 character per line.
iClones [41]	1,2,3	Min length 50 tokens, min block 20 tokens.	Min length 100 tokens, min block 20 tokens.
NiCad [110]	1,2,3	Min length 6 lines, blind identifier normalization, identifier abstraction, min 70% similarity.	Min length 15 lines, blind identifier normalization, identifier abstraction, min 70% similarity.
SimCad [136]	1,2,3	Greedy transformation, unicode support, min 6 lines.	Greedy transformation, unicode support, min 15 lines.
Simian [44]	1,2	Min length 6 lines, ignore identifiers and literals.	Min length 15 lines, ignore identifiers and literals.

9.2.1 BigCloneBench

Type-1 CCFinderX, CPD, iClones, NiCad and SimCad have perfect Type-1 recall. CtCompare and Simian also have excellent recall at 95%. Duplo has good recall at 89%. ConQat (67%) and Deckard (60%) fall behind the others.

Type-2 Only NiCad has perfect recall for the Type-2 clones. CCFinderX, ConQat, CPD and SimCad have excellent recall, all $\geq 90\%$. iClones maintains good recall at 82%. CtCompare, Duplo and Simian have decent recall in the 70%s, while Deckard has poor recall at 58%. Duplo performs well despite not supporting Type-2

Table 9.3: Benchmark Recall Measurements and Difference Per Clone Type

Tool	BigCloneBench						Mutation Framework			Difference		
	T1	T2	VST3	ST3	MT3	WT3/T4	T1	T2	VST3	ΔT1	ΔT2	ΔVST3
Duplo	89	74	46	8	0	0	38	0	0	51	74	46
CCFinderX	100	93	62	15	1	0	99	70	0	1	23	62
CPD	100	94	71	21	1	0	99	82	0	1	12	71
CtCompare	95	78	59	17	0	0	96	63	0	-1	15	59
Simian	95	78	53	13	0	0	81	90	0	14	-12	53
ConQat	67	90	73	33	1	0	91	90	86	-24	0	-13
Deckard	60	58	62	31	12	1	39	39	37	21	19	25
iClones	100	82	82	24	0	0	100	92	96	0	-10	-14
NiCad	100	100	100	95	1	0	100	100	100	0	0	0
SimCad	100	98	91	48	8	0	100	94	89	0	4	2

Table 9.4: BigCloneBench: Type-3 Recall

Tool	Syntactical Similarity Interval, x% to x+5%									
	50	55	60	65	70	75	80	85	90	95
ConQat	0	0	2	3	8	18	41	62	85	56
Deckard	10	14	15	18	24	28	39	34	52	75
iClones	0	0	1	2	5	19	36	39	75	91
NiCad	0	0	1	10	85	99	100	100	100	100
SimCad	5	7	13	16	23	45	46	77	85	99
CCFinderX	0	2	2	1	5	8	23	25	51	77
CPD	0	0	2	2	5	20	22	35	73	68
CtCompare	0	0	1	2	4	19	15	29	62	54
Duplo	0	0	0	0	1	3	7	17	36	60
Simian	0	1	0	1	2	5	22	23	45	63

normalizations. These Type-2 clones must contain a significant Type-1 region that Duplo detects and which is accepted by the *c-match*'s 70% coverage requirement.

Type-2 detection reduces to Type-1 detection after Type-2 normalizations are applied. Where Type-2 recall is lower than Type-1, we expect the tool is missing or struggling with particular Type-2 normalization(s). It is strange that ConQat has a very strong Type-2 recall, but the weakest Type-1 recall of these tools. Specifically, ConQat is not detecting Type-1 clones of a single particular functionality which contributed a large number of Type-1 clones to the benchmark. We were unable to determine why ConQat was missing these clones. Re-configuring ConQat for them made no difference. Ignoring these clones, ConQat has a Type-1 recall of 97%.

Type-3 For the Very-Strongly Type-3 clones, NiCad has perfect recall and SimCad has excellent recall (91%). iClones has good detection (82%), while ConQat (73%) and CPD (71%) have decent recall. The remaining tools have poor recall for these clones. NiCad has excellent (95%) detection for the Strongly Type-3 clones, while the other tools have poor detection. None perform well in the Moderately Type-3 or Weakly Type-3/Type-4 regions. Semantic-awareness may be needed to detect clones in these regions with good precision.

Many of the tools that do not formally support Type-3 clone detection (Table 9.2) have recall in the Very-Strongly Type-3 similarity region. In particular, CCFinderX and CPD have similar recall to the Type-3 detectors Deckard and ConQat. This is due to these tools detecting significant continuous Type-1 or Type-2 regions that cover at least 70% of a Type-3 reference clone. It is more desirable for a tool to include the Type-3 regions in its detection of these clones. Otherwise, the user has to manually recognize the larger Type-3 clone.

The Type-3 detectors have lower Type-3 recall than we expected. We therefore investigate Type-3 recall for finer grained syntactical similarity regions. Table 9.4 shows Type-3 recall per 5% interval of syntactical similarity. For example, the 75% interval includes all Type-3 clones with similarity in the range 75% (inclusive) to 80% (exclusive). We do not show recall below 50% similarity as none of the tools have noteworthy recall

in that range. We split the tools in this table based on formal Type-3 detection support. Only the tools above the splitting line feature Type-3 detection.

ConQat has good recall (85%) for Type-3 clones in the 90% interval. Oddly, it has significantly poorer recall for the more similar clones in the 95% interval, only 56%. It also has poor (62%) recall for the 85% interval. Its recall drops as expected for lower intervals. ConQat was configured with a 70% similarity threshold, and a maximum of 5 errors (Type-3 gaps). The error setting may be holding back ConQat's Type-3 detection. This setting has a strong impact on execution time, and we are already using a value larger than default (3). Deckard does not have high recall for any of the intervals, with only decent (75%) recall for the 95% interval. It has poor recall for the 85% and 90% intervals, despite being configured with its default 85% similarity threshold. iClones has excellent (91%) recall for the 95% interval, but only decent (75%) recall for the 90% interval, with very low recall for the lower intervals. iClones does not present a setting to increase Type-3 sensitivity. NiCad has perfect recall for Type-3 clones with at least 80% similarity. It has excellent recall for the 75% interval, and good recall for the 70% interval. Its recall drops sharply for intervals below 70%, which is expected as it was configured for a 70% similarity threshold. SimCad has excellent (99%) recall for the 95% interval, and good (85%) recall for the 90% interval. Its recall drops below 50% for the 80% interval and below. SimCad's SimHash sensitivity threshold was chosen empirically by the tool authors, so modification is not recommended. Deckard and SimCad are the only tools to have a notable, although small, recall for similarity intervals below 65%. Their detection strategies (AST, SimHash) may be more resilient to statement re-ordering.

Of the tools lacking formal Type-3 support, CCFinderX and CPD have the best recall for Type-3 clones, specifically for clones in the 90% and 95% intervals. Tools lacking Type-3 detection are able to detect Type-3 clones, by the *c-match*, when the clones contain a continuous Type-1 or Type-2 region covering at least 70% of the reference clone. These tools have poor Type-3 recall below 90% similarity, showing this scenario becomes rare for lower syntactical similarity.

Many of these tools have very strong Type-1 and Type-2 recall. Many perform well for the Very-Strongly Type-3 clones. Only NiCad performs well for the Strongly Type-3 clones. ConQat, Deckard and NiCad have Type-3 sensitivity configurations, which we left at their default values. Presumably these defaults were selected by the tool authors considering precision. The default values are what tool users are most likely to use, so these results reflect typical tool usage. The tools could be improved with increased Type-3 sensitivity, although this must be done while maintaining precision.

9.2.2 Mutation and Injection Framework

Type-1 Most of the tools have excellent Type-1 recall (≥90%), with perfect detection by iClones, NiCad and SimCad. The exception is Deckard and Duplo which have poor recall. Simian has good recall, but falls behind the other tools.

Type-2 ConQat, iClones, NiCad, SimCad and Simian have excellent recall, ≥90%. CPD has good recall, CCFinderX and CtCompare have decent recall, while Deckard again has poor recall. The Mutation Framework correctly identifies that Duplo does not support Type-2 normalizations and detection.

Type-3 The Mutation Framework correctly identifies that CCFinderX, CPD, CtCompare, Duplo and Simian do not support Type-3 detection. While they may be able to detect Type-1 or Type-2 regions within the Type-3 references, they are unable to capture the Type-3 regions. The Mutation Framework requires the tool to capture the clone-type specific edit it introduced to the synthesized clones. Most of the Type-3 tools perform well. iClones and NiCad have excellent recall for the Type-3 clones, both >95%. ConQat and SimCad also perform well, with >85%. Only Deckard performs poorly, with only 37% recall. Deckard performs uniformly poor across the clone types, suggesting that its weakness is not due to the handling of any particular clone-type specific difference. We believe this is due to it's outdated Java parser (Java-1.4 only).

9.2.3 Comparing the Benchmarks

Here we compare the BigCloneBench and Mutation Framework results for these tools (**RQ2**). These benchmarks use very different, but complementary, benchmarking strategies. The advantage of the Mutation Framework's synthetic benchmarking technique is it measures recall per clone type very precisely. Since it synthesizes its reference clones, it is able to determine if a tool successfully detected the clone-type specific regions of a reference clone. For example, the Mutation Framework will not accept a candidate clone as a match of a Type-3 reference clone if the candidate clone does not capture the Type-3 regions. Additionally, each reference clone contains only one type-specific change from the taxonomy, so clone-type-specific recall can be measured without bias due to features from the other clone types. The advantage of BigCloneBench's real-world benchmarking technique is it measures recall using real clones in real systems. The distribution of the clone types and features of the reference clones reflects what is found in real systems. BigCloneBench has complex clones that contain mixed features of the clone types. While the Mutation Framework can measure per clone-type recall more precisely, BigCloneBench shows how the tools perform for real clones. The advantages of both of these benchmarks are essential for understanding clone detection recall. We suggest that both benchmarking strategies are needed to fully evaluate the tools.

Since the benchmarks use very different methodologies, we consider them to agree if their recall measurement has an absolute difference no greater than 15%. This is the threshold we have used in previous work when comparing benchmarks [128]. We have highlighted in gray the cases where the frameworks disagree. The cases highlighted with light gray are cases where we expected disagreement due to the Mutation Framework's precise clone-type recall measurements. The cases highlighted with dark gray are the cases we did not anticipate.

The light gray highlighted cases are where the Mutation Framework has measured no recall, while Big-

CloneBench has measured a significant (>15%) recall. Neither benchmark is incorrect in these cases. Rather, each benchmark is telling us something different about the tools, as per their individual benchmarking advantages. These are cases where a tool does not formally support a clone type. For example, the light gray highlighted tools under the 'ΔVST3' header do not formally support Type-3 detection. The Mutation Framework requires the tools to detect the type-specific changes in the reference clones, so it measures no Type-3 recall for these tools. However, these tools may detect a significant (≥70%) Type-1 or Type-2 region in the Type-3 clones, so BigCloneBench measures a sizable recall. BigCloneBench tells us these tools can detect significant portions of the Type-3 clones, but the Mutation Framework tells us they cannot detect the portions containing the Type-3 differences. The quality of these tools' Type-3 detection is therefore very limited. They are not appropriate for automatic clone analysis, as automated tools will see these clones as Type-1 or Type-2, and provide incorrect analysis. These tools may also be inconvenient for use-cases involving manual inspection, as users will need to manually recognize the Type-3 features missed by the tool. These conclusions also hold for Duplo, which does not formally support Type-2 detection.

The cases highlighted with dark gray were not expected. The benchmarks disagree for Deckard for all clone types, with the Mutation Framework uniformly measuring lower recall. We believe this is due to limitations in Deckard's parser, which only supports the Java-1.4 specification. The Mutation Framework synthesized clones using code fragments from JDK6 and Apache Commons, both of which make significant use of generics (Java-1.5). Deckard may perform better for BigCloneBench if Java-1.5+ features are less commonly used. The Mutation Framework measures a significantly lower Type-2 recall for CCFinderX. We investigated the per mutation-operator recall of CCFinderX, and observed it only has low recall for Type-2 clones where a single instance of an identifier is renamed, but has good recall for the other Type-2 edit types. Perhaps this edit-type is rarer than the others in real-world clones, which is why BigCloneBench measures a higher recall. ConQat has significantly lower Type-1 recall measured by BigCloneBench. This is its poor detection of Type-1 clones from a particular functionality, as we mentioned earlier. We are not sure why Duplo's Type-1 recall is significantly lower with the Mutation Framework.

Overall, the benchmarks agree for most recall measurements. Ignoring the cases where the tools do not formally support a clone type (light-gray highlight), the benchmarks agree in 70% of the Type-1 cases, 78% of Type-2 cases, and 80% of Type-3 cases. Half of the cases of disagreement that are not related to clone type support are from Deckard, which is likely due to its parser limitations. This strong agreement between two very different benchmarking strategies builds our confidence that the measurements are accurate (**RQ2**).

9.3 Intra-Project vs. Inter-Project Performance

Traditionally, clone detectors have been designed to locate clones within a single software system. However, applications of clone detection extend to clones between distinct software systems. Intra and inter-project clones may have different properties, so the tools may have different recall for these contexts. In this section,

Table 9.5: BigCloneBench: Intra-Project vs Inter-Project Recall

Tool	Intra-Project Recall				Inter-Project Recall				Difference, $\Delta = (Intra - Inter)$			
	T1	T2	VST3	ST3	T1	T2	VST3	ST3	ΔT1	ΔT2	ΔVST3	ΔST3
Duplo	97	34	49	12	50	81	42	6	47	-47	7	6
CCFinderX	100	89	70	10	98	94	53	17	2	-5	17	-7
CPD	100	80	67	18	100	96	76	22	0	-16	-9	-4
CtCompare	96	38	52	14	88	85	66	19	8	-47	-14	-5
Simian	98	82	55	6	77	77	50	16	21	5	5	-10
ConQat	62	60	57	49	98	95	91	25	-36	-35	-34	24
Deckard	59	60	76	31	64	58	46	30	-5	2	30	1
iClones	100	57	84	33	100	86	78	20	0	-29	6	13
NiCad	100	100	100	99	100	100	100	93	0	0	0	6
SimCad	100	95	86	59	100	99	96	43	0	-4	-10	16

we compare the intra and inter-project recall of the tools using BigCloneBench (**RQ3**). For intra-project recall, we evaluate the tools only for the reference clone pairs whose code fragments are located in the same software system. This is the average recall of the tools in a traditional single-system clone detection scenario. For example, when a developer uses a clone detector to locate the clones in their software project. For inter-project recall, we consider only the reference clone pairs whose code fragments are located in different software systems. This is the average recall of the tools in a cross-project clone detection scenario. For example, when a company uses clone detection to locate code duplication across their products, or when a researcher studies code duplication across the open-source community.

Table 9.5 summarizes intra and inter-project recall per clone type, as well as their absolute difference. We do not include the MT3 and WT3/T4 categories as the tools have negligible recall for these clone categories. We consider a tool's difference in recall to be significant if it exceeds 15%, and these cases are highlighted in gray. We choose this threshold based on the distribution of the difference across these 40 per tool, per clone-type, cases. The average difference is ±13%. The average is pulled up by a handful of cases with considerable difference. Only 15 of the cases have a difference that meets or exceeds the average. These 15 cases have an average difference of ±28%, while the other 25 cases have a average difference of ±4%. We use the average difference of the 40 cases, rounded up to 15%, as our threshold. We believe we are being sufficiently cautious with this threshold, and are confident the cases exceeding the threshold are affected by differing properties of intra and inter-project clones.

Most of these tools exhibit significant differences between their intra-project and inter-project recall for at least one of the clone types. Only NiCad exhibits no significant differences between these clone contexts. ConQat has significantly different recall for four of the clone types. Generally, it has better recall for inter-project clones, although it has better intra-project recall for the ST3 clones. Duplo has considerably better intra-project recall for Type-1 clones, yet considerably better inter-project recall for Type-2 clones. CCFinderX, CPD, CtCompare, Deckard, iClones, SimCad and Simian have a difference in recall for only one of the clone types.

Many of the tools have significant differences in Type-2 recall, with better inter-project recall in each of these cases. This difference is considerable, with a -35% difference on average. Difference in Type-1 recall is not uniform, although the difference is considerable for ConQat and Duplo. Similarly for the VST3 recall, with ConQat and Deckard showing a considerable difference. The differences in ST3 recall are not as strong as for the other types. Perhaps intra and inter-project Type-3 clones in this similarity range have similar properties, or the tools are generally not sensitive to their differences.

While the participating tools were primarily designed for single-system clone detection, our findings show that they do not have a universal weakness in cross-project clone detection. Per clone type, some of the tools perform better for inter-project clones, some for intra-project clones, and in most cases no significant difference is found. Of the thirteen cases of significant difference, seven cases prefer inter-project recall, while six prefer intra-project recall. Five of these cases show significantly better inter-project recall for Type-2 clones, often by a considerable amount. These results can be used by users to decide which tool is best for their use-case.

9.4 Clone Capture Quality

While high recall is important, it is also important that a tool capture the clones in a way that is useful to a user's clone-related task (**RQ4**). We evaluated the recall of these tools using BigCloneBench and our coverage clone-matching metric (*c-match*). This metric accepts a candidate clone that covers 70% of a reference clone's source lines. This metric ignores any additional lines the tool reports beyond the boundaries of the reference clone. Since the reference clones of BigCloneBench are function clones, additional lines reported by the tool are external to the functions. These source lines may be from other functions surrounding the function clone, or class-definition syntax. Ideally, clone detection tools should respect function boundaries when reporting clones. Tools that report clones that extend beyond function boundaries have poorer clone capture quality because these clones have poorer usability.

Some primary use-cases of clone detection include refactoring, clone management and automatic clone analysis. Clones that extend beyond function boundaries, or that intersect multiple functions, do not imply any specific refactoring action [13, 109]. This requires the developer to manually trim and/or split the clone by functional boundaries before considering any refactoring tasks. In clone management, developers need to reason about a large number of clones, and the appropriate actions to prevent harm to software quality. Having to manually trim or split individual clones significantly increases the difficulty and cost of reasoning about a large number of clones.

Automatic clone analysis is used by development tools that aid clone refactoring and management, as well as by researchers who study software using clones. An example of automatic clone analysis is a development tool that monitors the changes a developer makes to a function, and recommends clones detected by a tool that the developer most likely wants to propagate the changes to. A clone analyzer reasons about clones

for the developer or researcher when there are too many clones for them to manually investigate. However, the analyzer may behave incorrectly, or produce poor results, when the clone spans multiple functions. The analysis algorithm and metrics likely assume that the input clones are contained within a single logical unit of code (e.g. function, block). In general, it is not useful for a tool to report a clone that extends beyond the boundaries of a function. Clones that span multiple functions are not meaningful since the order and position of functions in a class is not meaningful.

While respecting function boundaries is only one consideration of clone capture (i.e., reporting) quality, we have shown why it is important to the practical usability of the clones. In this section, we evaluate how well the tools respect function boundaries in their detection of the reference clones. We do this using two extensions of our *c-match* metric.

The **strict coverage metric**, *sc-match*, extends the *c-match* to also require the candidate clone to not extend more than l lines beyond the reference function clone's boundaries, as shown in (9.3). For this evaluation, we use a tolerance of 3 lines. This is a small enough extension beyond the boundaries of a clone that even automatic analysis could trim the candidate clone to the function boundaries without having to split the additional lines into an independent clone. It is small enough that a user should require minimal effort to visually recognize and ignore the extraneous lines past the function boundary.

$$\begin{aligned} sc\text{–}match(C,R,t,l) = c\text{–}match(C,R,t) \wedge \\ C.f_1.s \geq R.f_1.s - l \ \wedge C.f_2.s \geq R.f_2.s - l \wedge \\ C.f_1.e \leq R.f_1.e + l \ \wedge C.f_2.e \leq R.f_2.e + l \end{aligned} \tag{9.3}$$

Some tools may not respect function boundaries when reporting clones. Such a tool is a poor choice for automatic refactoring and analysis usages. However, as long as the target reference clone is clearly featured in the reported candidate clone, a user should be able to visually parse the reference clone with an acceptable increase in effort. To evaluate the tools from this perspective, we use the **featured coverage metric**, or *fc-match*. This metric requires the candidate clone to cover the reference clone, and for the covered portion of the reference clone to be a significant feature of the candidate clone. This is the *c-match* in both directions, as shown in (9.4). We decided a 70% coverage of the candidate clone is the minimum for the reference clone to be visually identifiable by the user without a significantly burdensome examination.

$$fc\text{–}match(C,R,t) = c\text{–}match(C,R,t) \wedge c\text{–}match(R,C,t) \tag{9.4}$$

If recall measured by the *sc-match* and the *c-match* is similar, then we know that the tool respects function boundaries when reporting the reference clones, and is therefore a good candidate for both manual and automatic refactoring and analysis use-cases. If recall by the *sc-match* is much lower than by the *c-match*, then the tool does not respect function boundaries. Such a tool is not appropriate for use-cases that use automatic analysis. However, if the tool has similar recall measured by the *fc-match* and *c-match*,

than the tool features the function reference clones in its detection of them. Such a tool is appropriate for use-cases that use manual analysis, although with some increased effort compared to a tool that respects function boundaries. A tool with significantly lower *sc-match* and *fc-match* than *c-match* recall neither respects function boundaries nor features the reference clones. Such a tool is likely burdensome to use for most use-cases.

We compare the tools' recall per clone type for the *c-match*, *sc-match* and *f-match* in Table 9.6. We include only the VST3 and ST3 Type-3 categories as the tools do not have appreciable recall for the other Type-3/4 categories. We observe trends in these results related to the clone types the tools support, so we organize the tools with those that formally support Type-3 detection above the splitting line. We consider the *sc-match* or *fc-match* recall to be similar to the *c-match* recall if the relative difference is no greater than 20%. We use a relative difference because the tools all have different base recall performances. We use a generous threshold to favor the tools in this evaluation. We highlight in gray these cases of similarity. A highlighted *sc-match* recall indicates the tool respects function boundaries for that clone type, while a highlighted *fc-match* indicates the tool features clones of that type when it detects them. The threshold indicates these conclusions hold for at least 80% of the reference clones the tool successfully detects by the *c-match*.

The results show that the Type-3 clone detection tools have exceptional respect for function boundaries. Except for Deckard, recall measured by the *sc-match* and *fc-match* is identical to that measured by the *c-match* for these tools. Deckard has only minor reduction in recall between the *c-match* and *sc-match*. In our experiences with Deckard, it occasionally has errors in its clone boundaries. However, the effect seems to be minimal with respect to the reference clones. The Type-3 clone detectors (ConQat, Deckard, iClones, NiCad, SimCad) have excellent clone capture quality with respect to function boundaries, and are good candidates for any manual and automatic clone analysis use-cases (**RQ4**).

Conversely, for most of the clone types, the tools lacking formal Type-3 detection capabilities do not respect function boundaries, nor strongly feature their detection of the reference clones. In most cases, particularly for the Type-1 and Type-2 clones, their recall by the *sc-match* and *fc-match* is significantly lower than by the *c-match*. The exception is for the Strongly Type-3 clones, and a couple other instances. These tools do not support Type-3 detection. Their recall for Type-3 clones is due to the detection of a significant (70%) Type-1 or Type-2 region within the Type-3 clones. Most of these tools are respecting function boundaries for their detection of the ST3 clones. This is not because these tools respect function boundaries in general, but because the gaps in these clones is bounding the Type-1 or Type-2 region detected by these tools to within the function boundaries. This is lost with the VST3, where there are fewer gaps, and therefore it is less likely the the gaps will bound these tools detection within the function boundaries. Similarly for Duplo, which does not formally support Type-2 detection. Outliers are CCFinderX and Simian, that feature only the VST3 reference clones in their detection. These results show that the tools lacking Type-3 detection (CCFinderX, CPD, CtCompare, Duplo, Simian) neither respect function boundaries nor

Table 9.6: BigCloneBench: Clone Capture Quality - Metric Comparison

Tool	T1			T2			VST3			ST3		
	C	SC	FC	C	SC	FC	C	SC	FC	C	SC	FC
Duplo	89	1	5	74	70	72	46	26	26	8	7	7
CCFinderX	100	8	15	93	10	72	62	33	51	15	9	10
CPD	100	14	21	94	19	20	71	38	55	21	17	17
CtCompare	95	1	3	78	3	4	59	25	27	17	15	15
Simian	95	12	19	78	51	52	53	25	47	13	11	11
ConQat	67	67	67	90	90	90	73	73	73	33	33	33
Deckard	60	59	59	58	58	58	62	57	57	31	25	26
iClones	100	100	100	82	82	82	82	82	82	24	24	24
NiCad	100	100	100	100	100	100	100	100	100	95	95	95
SimCad	100	100	100	98	98	98	91	91	91	48	48	48

feature the function clones in their detection of them. Therefore, these tools are not appropriate for automatic analysis, and they may be burdensome for use-cases with manual inspection (**RQ4**).

9.5 Threats to Validity

Alternate configurations of the tools may result in better or worse recall. Wang et al. [139] refer to this as the confounding configuration choice problem, and it is a challenge in all clone studies. We took steps to ensure the tool configurations were appropriate for our study. We used configurations that target the known properties of the benchmark, such as clone types and clone size. Otherwise, we referred to the defaults and recommendations of the tools with respect to our knowledge of the benchmarks. This is the process a user would use to configure a tool for their own system, so our results reflect what a user should expect to receive. We did not execute the tools for various settings until an optimal result is found, as it is not possible for users to do this in practice. For the Type-3 clone detectors, lowering their thresholds would allow them to detect more clones in BigCloneBench [65]. However, the tools would have poor precision for low similarity thresholds.

9.6 Conclusion

We introduced BigCloneBench [125] as a big data, varied and comprehensive clone benchmark for modern tools. In this study, we evaluated ten clone detection tools using BigCloneBench (**RQ1**), and compared these results against our Mutation Framework (**RQ2**). We found the tools have strong detection of Type-1 and Type-2 clones, as well as Type-3 clones with high syntactical similarity. Improvement is needed in the detection of Type-3 clones with lower syntactical similarity, as well as Type-4 clones, while maintaining high precision, which may require semantic awareness. These real-world and synthetic benchmarks have high agreement, so we are confident in their accuracy (**RQ2**). Since BigCloneBench contains both intra and inter-

project clones, we were able to evaluate the tools for these contexts. We found that while many of the tools have different recall for single-system and cross-project detection scenarios, neither context was universally favored by the tools (**RQ3**). Using multiple clone-matching metrics with BigCloneBench, we showed that only the Type-3 tools respect function boundaries when reporting clones (**RQ4**). Clones reported by the other tools may have poorer usability in refactoring and automatic clone analysis use-cases. With BigCloneBench and the Mutation Framework, we believe we have created a solid foundation for measuring the recall of clone detection tools.

Chapter 10

BigCloneEval

In order to make BigCloneBench more accessible, we introduce BigCloneEval, a framework for evaluating clone detection tools using BigCloneBench. It is based on the tool evaluation procedure we have used in our tool comparison studies [116, 122] (Chapter 9). BigCloneEval makes it very easy for users to evaluate and compare the recall of clone detection tools with BigCloneBench. The user does not have to write any evaluation code beyond configuring their candidate tools for execution and converting clone detection reports to a standard format. BigCloneEval handles the execution of the candidate clone detector for IJaDataset, including managing possible scalability constraints of the tool using deterministic input partitioning. BigCloneEval tracks the detected clones, and efficiently determines which of the reference clones in BigCloneBench the tool was able to detect. The evaluation experiment is highly configurable. The user can specify constraints on the reference clones considered when measuring recall, can customize the clone matching algorithm, or can provide their own clone matching algorithm by a plug-in architecture. BigCloneEval produces a tool evaluation report which summarizes recall per clone type, for both intra-project and inter-project clones, for different syntactical clone similarity regions, and for clones implementing different functionalities. The goal of BigCloneEval is to make BigCloneBench accessible to the community, and to provide a standard in tool evaluations with BigCloneBench.

This chapter is based upon our manuscript [129] "BigCloneEval: A Clone Detection Tool Evaluation Framework with BigCloneBench" published by myself and Chanchal K. Roy and in the Tool Demonstration Track of the International Conference on Software Maintenance and Evolution (2016). I was the lead author of this paper and study, under the supervision of my supervisor Chanchal K. Roy. The publication has been re-formatted for this thesis, with small modifications to better fit the thesis.

The remainder of this chapter is organized as follows. We describe the version of BigCloneBench included with BigCloneEval in Section 10.1. The framework is described in Section 10.2, including its commands, evaluation procedure, customizations, and evaluation report output. Limitations in the framework are discussed in Section 10.3, and the framework is compared to the related work in Section 10.4. The chapter is concluded in Section 10.5.

10.1 BigCloneBench - BigCloneEval Release

The BigCloneEval version of BigCloneBench contains clones mined for 43 distinct functionalities. As there is no consensus on the minimum syntactical similarity of a Type-3 clone, it is difficult to separate the Type-3 and Type-4 clone pairs that implement the same functionality. Instead, BigCloneEval separates the clones into four categories based on their syntactical similarity. We define **Very-Strongly Type-3 (VST3)** clones as those with a similarity in range 90% (inclusive) to 100%, **Strongly Type-3 (ST3)**: 70-90%, **Moderately Type-3 (MT3)**: 50-70%, and **Weakly Type-3 or Type-4 (WT3/4)**: 0-50%. Syntactical similarity is measured for each reference clone as the ratio of the lines or tokens a code fragment shares with another after Type-1 and Type-2 normalizations. Shared lines or tokens are identified by unix-diff [34]. We classify the clones into these categories using the smaller of their line and token-based clone similarity measures. Further details on BigCloneBench are found in Chapter 8.

10.2 Framework

BigCloneEval makes it easy to measure the recall of clone detection tools using BigCloneBench. It implements an experimental procedure similar to the one we have used in our previous clone detection tool evaluation experiments [116, 122]. BigCloneEval automates the major steps of the experiment, and allows the recall evaluation to be customized. It produces an extensive recall evaluation report that fully highlights the capabilities of a candidate clone detection tool.

BigCloneEval has four primary components. (1) The BigCloneBench database, which documents the reference clones of BigCloneBench. (2) IJaDataset, the inter-project Java repository containing the reference clones. (3) A tools database, which tracks the clone detection tools being evaluated by the framework, and their detected clones. (4) A set of command-line tools for interacting with the framework, including the registering of clone detection tools, performing clone detection for IJaDataset, importing the detected clones, and performing the recall evaluation experiments. Table 10.1 lists the commands, which we describe further in the following sections.

BigCloneEval is distributed as a git repository, so that users can easily pull updates. BigCloneBench and

Table 10.1: BigCloneEval Commands

Command	Description
registerTool	Registers a tool with the framework.
listTools	Lists the tool(s) registered with the framework.
deleteTool	Removes a tool, and its detected clones, from the framework.
partitionInput	Partitions a clone detection input given a maximum input size.
detectClones	Automates the execution of a tool for IJaDataset.
importClones	Imports a tool's detected clones into the framework.
clearClones	Removes the imported clones of a tool from the framework.
evaluateTool	Measures the recall of a tool and produces the tool evaluation report.

IJaDataset are downloaded separately, and added to the distribution. BigCloneEval uses fast and efficient embedded databases so that the user does not have to install and setup a database server. The BigCloneBench database [127] and IJaDataset [127] repository are very large, so BigCloneEval uses special versions of these that contain only the data and source files needed to perform the recall measurement, reducing their storage requirements.

10.2.1 Evaluation Procedure

The tool evaluation procedure is shown in Figure 10.1. First the clone detection tool is registered with the framework, which assigns it a unique tool ID. Next, the tool is executed for IJaDataset, and its detected clones are collected. As a speedup, the tool only needs to be executed for the files in IJaDataset that contain clones in BigCloneBench. Clone detection can be executed manually by the user, or the framework can automate this process, including overcoming possible scalability limits of the clone detection tool using deterministic input partitioning. Then, the detected clones are imported into the tools database for the given tool. Lastly, the tool is evaluated against the clones in BigCloneBench. The evaluation is highly configurable, and the output tool evaluation report summarizes the tool's recall per clone type, per syntactical similarity region and per functionality in BigCloneBench. These individual steps, the output, and the framework commands are detailed in the remaining sections.

10.2.2 Register Tool

The candidate clone detection tool is registered with the framework using the ***registerTool*** command, which requires the name of the tool and a description of its configuration for the experiment. These are stored in the database for reference, and a unique identifier is provided to the user for specifying this tool with the commands of the proceeding steps. The registered tools, their IDs, names and descriptions, can be listed using the ***listTools*** command. Tools can be removed from the framework using the ***deleteTool*** command.

10.2.3 Detect Clones

Next the user must execute their candidate tool for IJaDataset and collect the detected clones. IJaDataset is very large, and outside the scalability limits of most clone detection tools. However, the clone detection tools do not need to be executed for the entire IJaDataset, only for the files containing reference clones in BigCloneBench. We provide a reduced version of IJaDataset which contains only the relevant source files and is split into a number of smaller subsets for clone detection. There is one subset per functionality in BigCloneBench. Each functionality's subset includes all the files which contain a function tagged as a true or false positive of that functionality in the creation of BigCloneBench. Therefore each subset is a realistic subject system, containing both true and false positive clones. The tool must be executed for each subset of IJaDataset, and the clones collected. This is equivalent to executing the tool for the entire IJaDataset, in

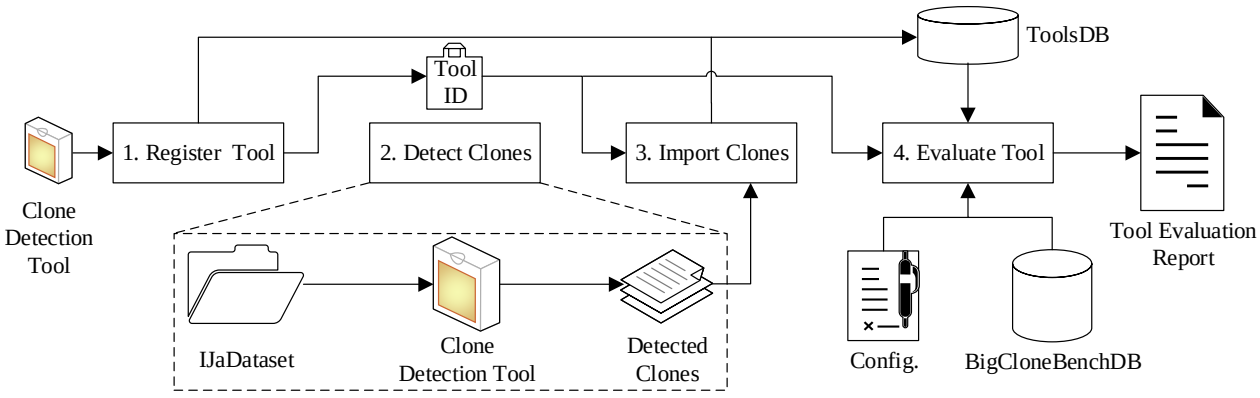


Figure 10.1: BigCloneEval Evaluation Procedure

terms of measuring recall for the reference clones.

A couple of these subsets may still be too large for some clone detection tools, specifically those that do not scale well in memory. This can be overcome using a deterministic input partitioning approach [84]. This involves partitioning the input and executing the tool for each unique pair of partitions. Partition size is chosen such that a pair of partitions does not exceed the scalability limits of the tool and available hardware. To perform deterministic input partitioning we provide a ***partitionInput*** command. This takes a directory of source files, a maximum input size in source files, and an output directory. Within the output directory it creates a subdirectory of source files for each unique pair of partitions given the maximum input size. Executing the tool for each subdirectory is equivalent to executing it for the original input.

While the user can perform the above manually with their clone detection tool, we also provide the ***detectClones*** command which automates the detection procedure. The user provides a script that configures and runs their tool, and the maximum input size considering their tool and available hardware, if required. The framework will automatically execute the tool for each subset of IJaDataset, using partitioning when needed, and collect the detected clones into a single output.

10.2.4 Import Clones

Now the user imports the clones detected by their clone detection tool into the tools database. This is done using the ***importClones*** command, which takes the ID of the registered tool and a file containing the clones to import. The clone file must list the clone pairs detected by the clone detection tool in a simple CSV format.

10.2.5 Evaluate Tool

The ***evaluateTool*** command is used to measure the recall of the clone detection tool, and produce its tool evaluation report. This command requires the ID of the registered tool to evaluate, whose detected clones have already been imported, and a file to output the recall measurements to. It iterates through each reference

clone in BigCloneBench and uses a clone matching algorithm to determine if the candidate tool was able to detect them. Recall is summarized for strategic subsets of the benchmark (e.g., per clone type) in the tool evaluation report (discussed further in Section 10.2.6). The user can configure the evaluation procedure with a number of constraints on the clones considered when measuring recall. They can also customize or provide their own clone matching algorithm. We describe these further in the following subsections. By default, a configuration matching our previous clone benchmark experiments is used [116, 122].

Reference Clone Selection

The user can specify a number of constraints on the reference clones considered when measuring recall. Users can select clones for consideration by minimum and maximum clone size as measured by language-tokens, pretty-printed source lines, and/or original source lines. These options can be used to measure recall for clones within particular clone size ranges. They are also useful for reducing bias when measuring and comparing the recall of multiple tools. Clone detection tools typically require at least a minimum clone size configuration, and most tools measure clone size by token or by source line (original or pretty-printed). By selecting a strict minimum and maximum clone size by each measure, the tools can be appropriately configured for BigCloneBench, and their recall results can be compared without bias due to clone size configuration. Users can also select reference clones by the total number of judges that have examined the code fragments of a reference clone, and their collective confidence in their judgment of those code fragments (the difference of true and false positive votes).

Clone Matching Algorithm

Recall is measured using a clone matching algorithm, which judges whether a reference clone in BigCloneBench is successfully detected by a candidate tool. BigCloneEval includes our coverage-based clone matcher, which we have used successfully in our previous work [116, 122], and is based on our *covers* metric. A code fragment f_1 covers code fragment f_2 if it intersects a ratio t of the source lines of f_2, as shown in Eq. 10.1, given that the code fragments are in the same source file. A reference clone R in BigCloneBench is considered detected by the candidate clone detector if there exists a candidate clone C reported by the candidate tool that satisfies the clone matcher. The coverage matcher is shown in Eq. 10.2, and requires the code fragments of C to cover the code fragments of R given a minimum coverage threshold t. Both orderings of the candidate clone's code fragments are tested. The coverage clone matcher is implemented as a database query over the tool's imported clones. Database indexes are used to make this query efficient, as the number of reference clones in BigCloneBench is very large.

$$covers(f_1, f_2, t) = \frac{min(f_1.e, f_2.e) - max(f_1.s, f_2.s) + 1}{f_2.e - f_2.s + 1} \geq t \tag{10.1}$$

$$c\text{–}match(C, R, t) = covers(C.f_1, R.f_1, t) \wedge covers(C.f_2, R.f_2, t) \tag{10.2}$$

The user can choose the coverage threshold of the coverage matcher (the default is 70%), as well as set a number of advanced configurations. The user can also provide their own custom clone matcher by a plug-in architecture. The user specifies the the name of the clone matcher and a configuration string. The clone matcher is discovered and configured at runtime. The existing coverage clone matcher can be used as a template by the user when implementing their own algorithm.

10.2.6 Tool Evaluation Report

The tool evaluation report summarizes the tool's recall performance for BigCloneBench given the configuration of the ***evaluateTool*** experiment. Recall is summarized per clone type, including the Type-3/Type-4 categories discussed in Section 10.1. Recall is also measured for different minimum syntactical similarity thresholds, as well as for different regions of syntactical similarity. Recall is summarized for all clones, for just the intra-project clones, and for just the inter-project clones. It is also summarized for all clones, and for each of the individual functionalities in BigCloneBench. The report also summarizes the reference clones of BigCloneBench considered given the configuration of the experiment (e.g., clone size). The report names the versions of BigCloneBench and BigCloneEval used to measure recall, as well as the configurations of the experiment, including the clone matcher, for future reference.

10.3 Limitations

BigCloneEval performs our clone detection tool recall evaluation procedure [116,122]. While it has a number of customization options, including allowing custom clone matching algorithms, it does not extend beyond this procedure. The framework is open-source, so users can adapt the procedure if needed. As well, the full BigCloneBench database is available for users who are developing novel research studies and evaluation procedures [127]. BigCloneEval does not measure clone detection precision. There is no existing methodology for measuring precision automatically, and is typically done by manual clone validation. BigCloneEval measures recall in terms of clone pairs, while some tools also report clones as clone classes. There is not a standard for measuring recall considering clone class reporting. It is an open topic we would like to explore in future work, and integrate into BigCloneEval.

10.4 Related Work

Bellon et al. [13] provide a benchmark of four thousand clones and a framework for evaluating clone detectors against this benchmark. Bellon's benchmark was built by manually validating a small fraction of the clones detected by participating tools in their benchmarking experiment [13]. Therefore, it is limited by the clone detection capabilities of its participating tools, which also introduces some biases [9]. Murakami et al. [93] extended Bellon's benchmarking by identifying the gap lines in Bellon's benchmark. Charpentier et al. [19]

re-examined some of the clone validation efforts in Bellon's Benchmark and found disagreement in the results when multiple judges are used. We previously found that Bellon's Benchmark may not be appropriate for evaluating modern clone detection tools [128]. In contrast, BigCloneBench is a much larger benchmark, and was built independently of the clone detection tools in order to avoid bias. We introduced the Mutation and Injection Framework, which automatically measures the recall of clone detection tools in a mutation-analysis procedure. Its synthetic benchmarking compliments the real-world benchmarking strategy used by BigCloneEval.

10.5 Conclusion

In this chapter, we introduced BigCloneEval, a framework for measuring the recall of clone detection tools using our BigCloneBench. BigCloneEval makes it very easy to perform clone detection tool benchmarking experiments with the reference clones in BigCloneBench. It gives the user flexibility over the configuration of the evaluation experiment, including the clone matcher used. Recall can be measured for both inter-project and intra-project clones, with recall summarized per clone type, per syntactical similarity range, and per functionality in the benchmark.

Part III

Large-Scale Clone Detection

In this part, we present our work on large-scale clone detection. Specifically, clone detection that can scale to inter-project source-code datasets on the order of hundreds of millions of lines of code. We performed two major studies in this area. In our first study, we use input partitioning and input shuffling with heuristics to scale the classical clone detection tools to large-scale, at the cost of an acceptable reduction in recall performance. In our second study, we present our large-scale clone detection tool, CloneWorks, which advances the state of the art in clone detection in terms of scalability and speed, while introducing our novel user-guided approach. The user-guided approach allows the user to customize their clone detection experiment to target any type or kind of clone, as per their scenario or use-case, including to pursue new varieties of clones.

In Chapter 11, we present study on scaling the classical (natively non-scalable) clone detection tools. We designed the Shuffling Framework, a methodology for reducing a large source dataset into a series of smaller subsets for clone detection. This successfully scales classical clone detectors to large scale at the cost of an acceptable loss of their native recall performance. The subsets are kept small enough for the tool to scale on average hardware without hitting memory or time constraints. We begin with our core Shuffling Framework, which partitions the dataset, and then randomly shuffles the source files into different partitions over a number of detection rounds. We then explore heuristics to improve the recall achieved within fewer detection experiments. We find the best scalability by building the subsets by shuffling together pairs of similar source files, and efficiently tracking those files which have been shuffled together previously to avoid repetition. While this framework successfully scales the tools, we find that a small cluster of workstation computers is necessary to achieve scalability in execution time. The Shuffling Framework enables us to take advantage of the desirable features and properties of existing tools while targeting inputs outside of their typical scalability range.

In Chapter 12, we present our large-scale clone detection tool, CloneWorks, which is designed for fast, scalable and user-guided clone detection experiments. CloneWorks detects clones using a simple Jaccard-based clone similarity metric, which represents code fragments as sets of code terms, and detects clones clones as code fragments that share a minimum overlap ratio. This clone detection metric is scaled in execution time using the sub-block filtering optimization and clone indexing techniques from our previous works [116, 126]. We implement this technique for speed at the cost of high memory usage, and then scale within limited memory using an input partitioning technique based on our Shuffling Framework. We achieve a user-guided approach using our input converter, which allows the user to customize how their code fragments are transformed into code-term sets. Specifically, the user chooses the source-level transformations, term splitting and term-level transformations applied the code fragments, including a plug-in architecture for further customization. This enables the user to target any type of clone, or even new kinds of clones such as API clones. We design a parallel and efficient architecture that enables best-in-class execution time and scalability, even on limited hardware. We perform a large-scale tool evaluation experiment that measures recall, precision, scalability and execution time for CloneWorks and the competing tools. Through scenarios and case-studies, we demonstrate the user-guided aspect of CloneWorks, including the validation of over 15K

detected clone pairs from targeted clone detection. CloneWorks will enable researchers to pursue new areas of large-scale clone detection research.

Chapter 11

Large-Scale Clone Detection using the Classical Detectors

Scalable clone detection is amongst the most active topics in the clone community. One of its primary goals is the creation of clone corpora from ultra large inter-project datasets that often contain on the order of thousands of open-source systems. However, building scalable tools is challenging and it is often impossible to use existing state of the art tools for big data analysis, except for emerging tools that are built for extreme scalability. Reasons for their failure include insufficient memory, impractical computation time, and/or limitations in their underlying algorithms.

In this chapter, we develop and evaluate a scalability heuristic we call the shuffling framework [64] [120]. Our technique allows classical clone detection tools (i.e., those not specifically designed for big data) to be scaled to big data on standard workstation-class hardware without modification. The framework achieves scalability by executing the classical tool for subsets of the dataset. The subset size is kept small enough that the tool does not encounter scalability issues when executed on a standard workstation. The subsets are chosen by a non-deterministic process that "shuffles" the dataset's files into inputs for the classical tool. Using the tools in their original state ensures that their native precision and detection characteristics are maintained when executed through the framework. By executing the tool for a sufficient number of subsets an acceptable ratio of the tool's native recall is achieved for a dataset outside of its native scalability. The key to the performance of the framework is the design of the non-deterministic strategies used to choose the subsets.

This research is motivated by the richness of inter-project clone corpora for software mining experiments and applications. Inter-project datasets of interest include public open-source repositories (e.g. SourceForge and GitHub) as well as private corporate repositories. Clone corpora may be mined to study developer behavior both globally (e.g., across open-source repositories) or within an organization (e.g., across a company's private repository). They can be used to discover frequently re-implemented functionalities that should be extracted into new software libraries to remove duplicated engineering costs. A corpus may also be used as a basis for Internet-scale clone search [61], which has applications including API recommendation and usage support. Scalability in detection is achieved using either novel scalable detection techniques (general or domain specific), or mixing classical approaches with scalability heuristics.

One of our goals is to allow classical tools to contribute towards inter-project clone corpora (e.g., [61]). It is not sufficient to only consult scalable clone detectors when creating a clone corpus as classical tools have their own unique strengths and detection characteristics. While general-purpose scalable detection techniques exist in the literature, most have not been publicly released as user-friendly tools. Additionally, scalable tools are still novel and their recall and precision have not been proven. Classical tools have matured and there is more understanding and confidence in their abilities and detection quality. In order to build a truly comprehensive inter-project clone corpus, a variety of detection tools need to be consulted, including both scalable and classical tools.

We propose a shuffling framework based on input partitioning. This strategy completely partitions the dataset into disjoint subsets. The tool is then executed for each subset to locate the clones within these partitions. Since it is likely that files containing clones will be assigned to different partitions, the contents of the partitions are randomly shuffled over a number of rounds. Rounds are executed until either the framework user has met their time constraints, or when the cost of executing an additional round exceeds the expected benefit (as judged from the previous rounds) in terms of the number of new clones detected. This strategy relies upon randomization to shuffle clones together. It assumes that while a large number of clones remain to be found, there is a good chance random selection will shuffle files containing clones into the same partitions. The technique should reach a point of diminishing returns when a significant portion of the tool's native recall has been found. This shuffling strategy is computational cheap, and the creation of the subsets is negligible compared to the cost of executing the clone detection tool.

We evaluate this partition strategy for the ultra large inter-project dataset IJaDataset 2.0 [60] using a selection of classical clone detection tools, including Deckard [53], NiCad [105], iClones [41], Simian [44], SimCad [134] and CCFinderX [58]. We measure the framework's detection performance as the ratio of a tool's native recall that it is able to capture. This study reports our observations and the challenges faced in executing our framework for these tools and dataset. In order to gauge the expected performance of the framework for these tools, we also executed it for standard size datasets which allowed us to compare clone detection with and without the framework. We developed and evaluated a heuristic for estimating framework performance when the clone output was too large to process on available hardware.

From our performance observations we identified the strengths and deficiencies of the partitioning approach. Generating the subsets by randomly partitioning the dataset over a number of shuffling rounds is computationally inexpensive, and ensures the tool is exposed to every file in the dataset. However, we found a large number of rounds was required to obtain an acceptable ratio of a tool's native recall. We identified two attributes of random partitioning that limits its performance.

First, when shuffling the partitions the framework does not consider which files have been shuffled together previously. It is possible, especially as more rounds are executed, that pairs of files will be randomly shuffled together more than once. Executing the tool for the same pairs of files repeatedly costs computation time and resources without discovering new clones (i.e., improving recall). Computing rounds of partitions that

never shuffles the same files together more than once is neither simple nor cheap. However, minimizing the reshuffling of the same files together repeatedly would improve the performance of the framework.

Second, this strategy does not consider the similarity of the files it shuffles together. A significant portion of a clone detector's computation time is spent searching for clones between files that do not contain clones. Shuffling together only those files that contain clones would reduce the amount of wasted time. Of course, determining if two files contain a clone has the same cost as clone detection. However, a cheap (i.e., $O(n)$, where n is the combined length of the files) heuristic to estimate if two files contain enough similar code to *possibly* contain a clone could be used to prevent files too dissimilar to contain a clone (as judged by the tool) from being shuffled into the same subsets.

Using these observations, we improved our framework's subset generation and file shuffling strategy. Specifically, we explored methods of efficiently tracking seen file pairs, and efficient heuristics to measure source file similarity. By tracking the seen file pairs, we can guarantee each new subset contains a minimum number of new detection experiences. By measuring file similarity, we can avoid shuffling together files that are unlikely to contain a clone as judged by the specific classic tool. The goal of these two heuristics is to maximize the number of clones found per subset the tool is executed for, which minimizes the number of subsets needed to obtain an acceptable ratio of the tool's native recall. This improves the scalability of our framework. We incrementally introduced these heuristics to the framework, and measured their performance in an experiment mimicking a real big data scenario.

Using our findings of the computational cost and recall performance of the added heuristics, we specify a final shuffling algorithm which merged the best features of the partitioning method and the heuristics. We used this final version of the shuffling algorithm to analyze IJaDataset. We compared the improved algorithm against the original core algorithm (the original shuffling framework [120] [64]). While the heuristics increased the cost of generating the subsets of the dataset to analyze, it greatly reduced the number of subsets the classical tool needed to be executed for to achieve a satisfactory ratio of its native recall, while of course retaining the tool's original precision.

In summary, this work answers the following research questions:

RQ#1 What is the accuracy of our heuristic for measuring the recall performance of the shuffling framework?

RQ#2 What is the expected recall performance of the core shuffling framework for these selected clone detection tools with respect to their native recall performance.

RQ#3 Is our shuffling framework successful in scaling classical detection tools to big data?

RQ#4 By observing the behavior of the shuffling framework, can we modify it to improve its recall performance in terms of recall and execution time?

RQ#5 Does the improved shuffling framework perform better for big data?

This chapter is based upon our manuscript [126] "Big Data Clone Detection Using Classical Detectors: An Exploratory Study" published by myself, Iman Keivanloo and Chanchal K. Roy in the Journal of Software: Evolution and Process (2015), ©2014 John Wiley & Sons, Ltd. I was the lead author of this paper and study, under the supervision of my supervisor Chanchal K. Roy and Iman Keivanloo. The publication has been re-formatted for this thesis, with modifications to better fit the thesis.

The remainder of this chapter is organized as follows. We begin with a short survey of related work in Section 11.1. The procedure of our core shuffling framework as proposed in previous work [64] is outlined in Section 11.2. Section 11.3 overviews our experimental set-up and defines our metrics, including the recall evaluation heuristic. We evaluate the expected performance of our core algorithm in the preliminary experiments detailed in Section 11.4. Section 11.5 discusses our experiences in applying our core shuffling framework to big data (IJaDataset), and reports our observations regarding the framework's clone detection performance. In Section 11.6 we analyze the performance and deficiencies of our core shuffling algorithm. In Section 11.7 we develop shuffling heuristics to address the deficiencies in the core approach, and incrementally integrate them into the core shuffling algorithm. Using a test dataset (a sample of IJaDataset) we measure the effectiveness of these improvements versus the costs the heuristics added to the shuffling algorithm. From these experiments, we specify an improved shuffling framework in Section 11.8. In Section 11.9, we revisit IJaDataset with the improved framework and compare our experiences against the core framework in terms of recall performance and tool scalability improvements. We conclude this research in Section 11.10.

11.1 Related Work

Scalable clone detection research can be summarized as five unique approaches: (1) deterministic novel general purpose detection (e.g., [73]), (2) deterministic novel domain-specific approaches (e.g., [48]), (3) deterministic approaches for achieving scalability by altering available tools (e.g., [114]), (4) deterministic approaches for achieving scalability using an available clone detection tool as is (e.g., [84]), and (5) nondeterministic approaches for scaling an available tool (e.g., [64]). A variety of use cases can be addressed using each family based on their unique features. Our Shuffling Framework is an implementation of approach (5).

Deterministic novel general purpose approaches, (1), are designed specifically for scalability. A number of such techniques for big data have been explored in clone literature, however public tool availability remains rare. Approaches that achieve scalability on a single machine may require compromises in granularity, recall and/or precision in order to reduce computational complexity or the clone search space. Other approaches achieve scalability by targeting scalable hardware, such as cloud-based computing clusters, which can be costly to purchase or rent.

Deterministic novel domain specific approaches, (2), achieve scalability by optimization for a particular use case. By exploiting domain knowledge of a particular use case, computational complexity can be lowered without significant compromise to detection features, recall or precision. However, the approach's

performance is strongly specific to its domain of study. The approach may be ineffective in other use cases.

Existing classical tools may be modified for scalability, (3). These approaches exploit the proven existing clone detection technique with modifications to improve its scalability. For example, an existing tool may be modified to distribute its computation. Or an heuristic may be used to reduce the search space the classical approach must be executed for. Improving the scalability of an existing tool may scale its hardware requirements, or reduce its recall and/or precision. To implement such an approach, the original tool's source code and expert knowledge of its implementation is required. Many tools are closed source, or are released without extensive design/engineering documentation.

Methods (4) and (5) use classical tools as-is and scale them to big data. These approaches exploit the known detection characteristics, recall and precision of available tools. Many classical tools are available, and users are confident in their abilities and correctness due to their widespread use in clone research. By not requiring modification to the tools, these approaches can scale closed-source tools. While open-source tools might be modifiable for increased scalability, this requires expert knowledge in their algorithms and implementations. Our Shuffling Framework exploits the non-deterministic method, (5).

Deterministic methods of scaling classical tools without modification, (4), is the most similar approach to our Shuffling Framework. An implementation of the deterministic approach (e.g., [84]) begins by partitioning the input dataset into disjoint subsets half the size manageable by the classical tool on workstation hardware. The tool is then executed for each pair of these subsets. If the tool's input scalability limit is $\frac{1}{n}$th of the big dataset, then the deterministic method will partition the input into $2n$ disjoint subsets, and execute the tool for $n(2n-1) = O(n^2)$ subset pairs. This strategy maintains a classical tool's native recall and precision while scaling it to big data by deterministic exposure of the tool to every file and pair of files in the dataset. However, for a dataset containing thousands of software systems, the deterministic method may require several weeks of execution time with a classical tool on a single workstation. Therefore, to achieve scalability in execution time, the user must distribute the analysis of the subset pairs across a large cluster of workstations.

Our Shuffling Framework aims to scale classical tools on a single workstation, or on a (small) handful of available workstations, without modifications to the tool. It executes the tool for some number of manageable subsets of the dataset. The subsets are chosen by a non-deterministic (random) process, meaning that the tool is not exposed to every file pair in the dataset. The approach relies upon randomization to allow an acceptable ratio of a tool's native recall to be achieved in a manageable number of subsets, at least far fewer than required by the deterministic method (i.e., approach 4). The probability of files containing clones ending up in the same subset is higher when a large ratio of a tool's native recall remains to be found. This approach maintains the classical tool's detection characteristics and precision, but sacrifices its recall to achieve scalability on affordable hardware. In this research, we develop and investigate non-deterministic methods of choosing these subsets as to reduce the number of subsets needed to achieve an acceptable ratio of a tool's native recall.

There are a few recent and similar studies to our research. Ishihara et al. [48] exploited inter-project scalable clone detection to locate commonly used functionalities within 13K open source projects in order to generate a seed for future APIs and libraries. Schwarz et al. [117] studied cloning between 3K Smalltalk projects to deploy a database of clones which can be queried. Ossher et al. [96] observed cloning at the file level using coarse-grained clone detection heuristics. Common to all these studies, the detection approach is customized and optimized considering the research objectives and requirements. This is contrary to our research where we show that a clone dataset can be generated using available clone detection tools by copping with the scalability issue without altering the tools.

11.2 The Core Shuffling Framework

The shuffling framework allows clone detection tools not designed for extreme scalability to scale to ultra large datasets without modification on standard hardware while achieving an acceptable overall recall and retaining the tool's original precision. Summarized below is our core Shuffling Framework approach as proposed in our earlier work [64]. We begin this research with the analysis of the core approach's performance for ultra-large datasets. We discuss improvements to this approach starting in Section 11.8. The core framework executes the following procedure:

1. The source files of the dataset are randomly partitioned into n disjoint subsets of equal size. Subset size is chosen such that the clone detection tool can handle a single subset within a single execution on standard hardware without encountering scalability difficulties.

2. Each subset is searched independently by the clone detection tool. This can be done sequentially, in parallel, or distributed over independent computers.

3. The detected clone pairs are merged into a clone repository.

4. Steps (1) through (3) are repeated for r rounds. Multiple rounds are required as a single round achieves limited recall. There is a high chance that cloned contents are assigned to disjoint subsets. Since the rounds are independent, they may be executed sequentially or in parallel on common or independent computing resources.

The framework achieves scalability by partitioning the dataset into subsets individually manageable by the clone detection tool. The tool's recall is recovered by repeating detection after shuffling the partition contents. The goal of this non-deterministic approach is to achieve an acceptable fraction of the tool's native recall within a manageable number of rounds ($O(nr)$ tool executions).

To use this framework, the user must select an appropriate subset size for their clone detection tool. Factors affecting this choice include how the tool's memory requirements, computation time, and algorithmic complexity scale with input size. Some tools may also have inherit input size limitations in their algorithms and data structures.

A number of rounds to execute must also be chosen. The more rounds executed, the closer the framework will come to the tool's native recall. However, the number of rounds executed must be manageable within available time and computing resource constraints. Preferably, rounds should be executed until the number of new clones found (i.e., discovered in the most previous rounds) is no longer worth the additional computation time. This decision depends on the individual use case.

Clones detected in each subset are added to a single clone repository. A clone may be detected in multiple rounds if its files are randomly shuffled into the same partitions in multiple rounds. Therefore the clone repository must handle the insertion of duplicate clones by retaining only one copy of the clone. In our implementation of the framework we used a hash-based set as a clone repository. This provides amortized O(1) clone insertion and lookup. Our clone pair equivalence function is defined to ignore code fragment order, so the set will only retain one copy of a clone pair even if the code fragment order is reversed.

11.3 Study Setup - The Corpus, Environment, Tools and Measures

11.3.1 Corpus - IJaDataset 2.0

For our experiment we used the second version of IJaDataset, which was constructed using raw data crawled in 2012 [60]. The dataset covers source code from approximately 25,000 open source Java projects. This new version of the dataset contains up-to-date source code and is two times larger than the first version, which we used in our earlier studies [64]. The dataset is based on source files mined from SourceForge and Google Code in 2012. The dataset includes nearly 3 million Java source files spanning 356 million lines of code (LOC). The dataset is publicly available [4].

Of the 3 million files in IJaDataset, 6238 are greater than 2000 lines in length. While these make up an insignificant portion of the dataset, they may contribute considerably to a clone detection tool's execution time. For this reason we consider these files as outliers of the dataset and omitted them from the experiments.

11.3.2 Hardware

Our framework aims to scale classical tools to ultra large datasets using **standard hardware**. Clone detection in big data is mostly of interest to researchers and professional developers. For this reason we used consumer-grade workstation-class desktop computers as our target for standard hardware. We expect such machines to have 4 or more processing threads on a modern CPU architecture (e.g., Intel Core i5) and 8-32GB of system memory. These machines should store active data on either a performance hard drive or, ideally, a solid state drive. At the time of publication, machines meeting these specifications cost approximately $800-$1500 USD.

The first IJaDataset experiment (Section 11.5) was executed in a distributed fashion on computing instances provided by the Bugaboo cluster of the Western Canada Research Grid (WestGrid) and Amazon

EC2. These instances meet our definition of standard hardware, and multiple were exploited in order to complete this study in a limited time-frame. The average instance included a 2.66GHz quad-core processor, 12GB of memory, and two 10,000RPM hard drives in raid0. All other experiments were executed on our local hardware, which includes a 3.6GHz quad core processor, 16GB of memory, and a single consumer-grade solid state drive. For particularly demanding analysis of the experiment's results (e.g., gold dataset creation and performance measurement), an EC2 instance with 64GB of memory was utilized. This extraordinary instance was never used for steps of the shuffling framework, only for analysis of the framework's performance.

Upon completion of the first IJaDataset experiment, we realized that traditional hard disk drives are the bottleneck to the framework. Subset creation and clone detection was considerably faster on our local machine using a consumer-grade solid state drive. For the experiments using standard hard disk drives we include estimates of what the execution time would have been on our local hardware based upon our findings with later experiments.

11.3.3 Clone Detection Tools

For this study, we explored six clone detection tools. Being freely available and supporting Java source code were our major deciding factors. Table 11.1 summarizes our selected tools and their chosen configurations. When possible, we preferred the tools' default settings. We used the same tool versions and configurations across all experiments.

11.3.4 Measures

The performance of our framework is measured as total recall: the ratio of the clones from the target tool's gold standard that the framework is able to find. The gold standard is the clones the target tool finds when run as is (i.e., without our framework). For the application of the shuffling framework for r rounds and n subsets, total recall is calculated using Eq. 11.1.

$$tr(r,n) = \frac{\left|(\bigcup_{i=1}^{r}(\bigcup_{j=1}^{n} detected\ clone\ pairs(i,j))) \cap (clone\ pairs\ in\ gold\ standard)\right|}{|(clone\ pairs\ in\ gold\ standard)|} \tag{11.1}$$

The numerator is the number of clone pairs in the tool's gold standard that it detected when executed through the framework. The set on the left of the set intersection is the set of unique clone pairs detected by the tool using the framework. The first union iterates over each round of the framework, while the second iterates over each subset in a round. $detected\,clone\,pairs(i,j)$ is the set of clone pairs detected in subset j of round i. The denominator is the number of unique clones in the gold standard. As this metric considers clone pairs, we also refer to it as clone recall or clone pair recall. It measures the ratio of the tool's native recall that the framework achieved.

We measure performance in terms of clone pairs instead of clone classes for a number of reasons. All tools either support clone pairs as output, or their clone class output can be simply converted into clone pairs. For

Table 11.1: Tool Configurations

Classical Subject Tools	Configurations
Deckard [53] (version 1.2.3)	Minimum fragment size of 50 tokens, and a sliding window of 5 tokens. Minimum 90% clone similarity (tree-based metric).
NiCad [105] (version 3.4)	Normalized fragment size of 10-2500 lines and minimum 70% clone similarity (line-based metric).
iClones [41] (version 0.1.2)	Minimum clone fragment size of 100 tokens and minimum cloned block size of 20 tokens.
Simian [44] (version 2.3.33)	Code fragment sizes of 6 lines or greater, no identifier or literal renaming.
SimCad [134] (version 2.1)	Detection of clone pairs of all types after consistent identifier normalizer.
CCFinder [58] (version 10.2.7.4)	Minimum fragment size of 50 tokens, with a minimum unique token type of 12.

tools that only report clone pairs, it is not trivial to convert their output to clone classes. It would either require modification to the tool, or the implementation of a clone clustering algorithm that uses the same decision logic and clone metrics as the tool. Disabling clone clustering in tools where it is an option may reduce their computation time and memory requirements. We disabled clone class output when possible to improve the native scalability of the tools.

Finally, considering clone pairs makes the calculation of total recall much more efficient. The clone reports from a tool executed by the framework can be merged into a hash set. The complexity of determining if a clone pair in the gold standard has been detected is then O(1). Since the number of clones these tools detect in IJaDataset is very large, the O(1) complexity is essential. This way, the evaluation of total recall is linear with respect to the size of the gold standard.

Heuristic-Based Total Recall Measurement

In our experience, a clone detector's output for ultra large datasets may be too large for the calculation of total recall in a reasonable time frame, even when extraordinary hardware is utilized (e.g., 244GB of RAM). Specifically, we experienced this when attempting to measure total recall for the framework's evaluation of IJaDataset using Simian. For this reason, a heuristic was devised to estimate total recall using limited time and resources. This heuristic estimates total recall by measuring the ratio of the cloned fragments, rather than clone pairs, from the gold standard that are found using the framework. Heuristic recall is measured using Eq. 11.2. The notation is the same as Eq. 11.1, except for cloned code fragments. As this metric considers cloned fragments, we also referred to it as cloned fragment recall or fragment recall.

$$hr(r,n) = \frac{\left|(\bigcup_{i=1}^{r}(\bigcup_{j=1}^{n} \mathit{detected\ cloned\ fragments}(i,j))) \cap (\mathit{cloned\ fragments\ in\ gold})\right|}{|(\mathit{cloned\ fragments\ in\ gold})|} \tag{11.2}$$

This heuristic is based on the assumption that if two cloned fragments of a clone pair have been found by our approach, then there is a good chance that the clone has also been detected, or that the clone could be recovered by applying the transitive property to all found clone pairs. For example, if fragments $f1$, $f2$ and $f3$ have been found in clone pairs $(f1, f2)$ and $(f2, f3)$ then the missed clone pair $(f1, f3)$ can be recovered. A caveat of this approach is that while it holds true for all clones of types 1 and 2, it does not for all type 3 clones.

Evaluation of our Heuristic-Based Recall Measure

In this study, we tested the assumptions of our heuristic-based recall measurement. We searched JDK1.7 using NiCad, Simian and Deckard both as they are and with our shuffling framework. The framework was parameterized to evaluate the dataset for 15 subsets over 30 rounds. Figure 11.1 compares the total recall and heuristic recall for the tools after each round. For NiCad and Simian, the transitive property was applied to recover additional clones. Recovered recall was then evaluated as in Eq. 11.3 by including the recovered clones per round as part of the tool's detected clones. Recovered recall was not evaluated for Deckard due to the size of its output.

$$rr(r, n) = \frac{|((detected\,clone\,pairs) \cup (recovered\,clone\,pairs)) \cap (clone\,pairs\,in\,gold)|}{|clone\,pairs\,in\,gold\,standard|} \tag{11.3}$$

As can be seen from these experiments, heuristic recall over estimates the total recall but follows a similar trend. Both show logarithmic growth in recall across the rounds. Cloned fragment (heuristic) recall starts higher, but has a slower growth across the rounds. The recovered recall performance for NiCad and Simian show the correctness of the heuristic. For NiCad the recovered recall approximately matches the heuristic recall. For Simian the recovered recall approaches heuristic recall after half of the rounds have been executed. This shows us our heuristic is effective in estimating the recall of our shuffling framework (**RQ#1**).

In this study we applied the transitive property naively. We assumed it held for all type 3 clones. This means we "recovered" false positive clones in the cases where transitivity did not hold between type 3 clones. However, these false positives do not affect the measure of recovered recall. Therefore, these results represent the ideal case where the recovery method successfully recovers all transitive clone pairs. In practice, a recovery technique would need to check that transitivity held before applying it to type 3 clones. It may not be possible to implement an efficient check that accepts all true positive transitive clones and rejects all false positive transitive clones. We used transitive clone recovery only in this evaluation of the heuristic recall measure. Creating an efficient transitive check with high recall (accepts most true positive transitive clones) and precision (rejects most false positive transitive clones) is a topic of future work. In this paper we use our total recall and heuristic recall measurements to comment on if a transitive clone recovery method is worth perusing in future work.

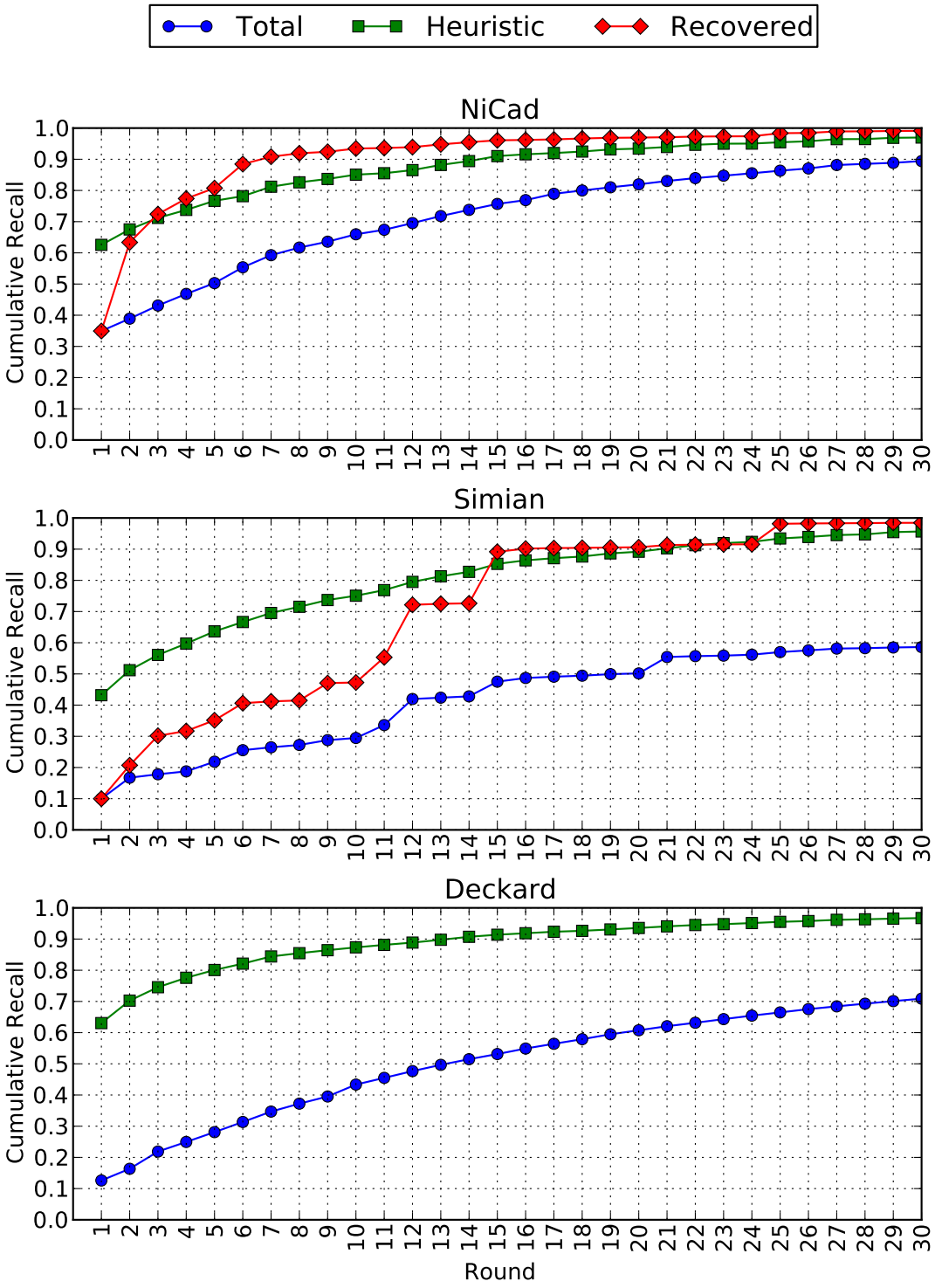


Figure 11.1: Comparison of the recall estimation approaches

11.4 Preliminary Experiments

We used the shuffling framework to evaluate three regular size subjects systems. This allowed us to evaluate the systems with the tools both natively (their gold standard) and with the framework. The goal of this experiment was to observe the expected performance of the framework for the six selected tools (**RQ#2**). We chose JHotDraw (20KLOC - 285 files), ArgoUML (190KLOC - 1845 files) and JDK1.7 (900KLOC - 6916 files) as our regular sized systems. The framework was parameterized for 15 random subsets and 30 detection rounds.

The framework's total recall performance for each tool's detection of JHotDraw54b1 is shown in Figure 11.2, ArgoUML is shown in Figure 11.3, and of JDK1.7 in Figure 11.4. The legends of these graphs specify the gold standard size (number of clones) for each tool. The framework performed very well with NiCad, iClones, and CCFinderX, obtaining a high total recall after 30 rounds. It struggled more for Deckard, and performed poorly with Simian for JDK1.7. Total recall started and ended lower for JDK1.7, but increased faster than for ArgoUML and JHotDraw, likely due to the differences in the sizes of the two systems (and gold standards). CCFinderX is omitted from the JDK1.7 experiment due to failure during detection.

In all cases we see approximately logarithmic growth in total recall across the rounds. As total recall becomes larger, the increase in total recall from each round decreases. This is expected, as the smaller the ratio of a tool's native recall that is left to be found, the lower the probability the files containing these undetected clones will be shuffled into the same subset. We saw the same trend with heuristic and recovered recall in the heuristic study, Section 11.3.4.

An observation from this experiment is that the larger the gold standard the lower the total recall obtained by the framework across the same number of rounds and subsets. This is seen here for both variation in detection tools and subject system size. The exception being Simian, for which the framework achieves a lower total recall than for tools with larger gold standards. Perhaps Simian has better precision for smaller datasets, and is therefore not finding the false positives in its gold standard, leading to a lowered total recall.

These results indicate that the framework can achieve an acceptable ratio (>70%) of a tools native recall given an acceptable number of rounds. The plots here show the expected framework performance for the tools, which answers **RQ#2**.

11.5 Motivating Study - IJaDataset

In this experiment, the clone detection tools were executed through the core shuffling framework to evaluate IJaDataset 2.0. This experiment was used to evaluate the performance and feasibility of the shuffling framework for clone detection in big data using classical tools (**RQ#3**).

Using a rented Amazon EC2 instance with 64GB of memory and 10,000 IOPS, we were able to obtain Simian's gold standard for IJaDataset. This allowed us to compare native versus framework recall in a big

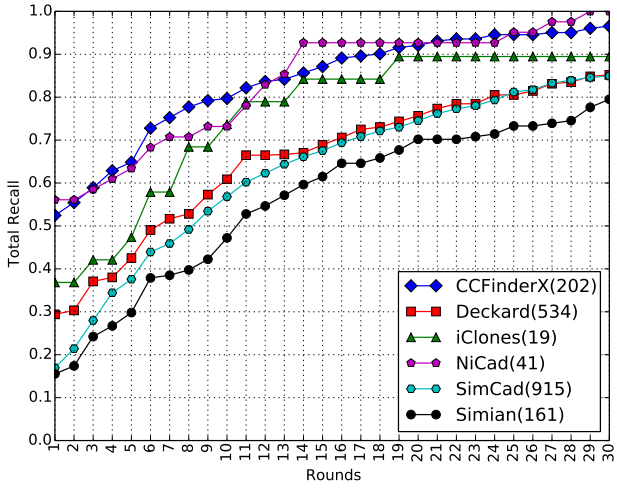


Figure 11.2: Preliminary Experiment - JHotDraw54b1

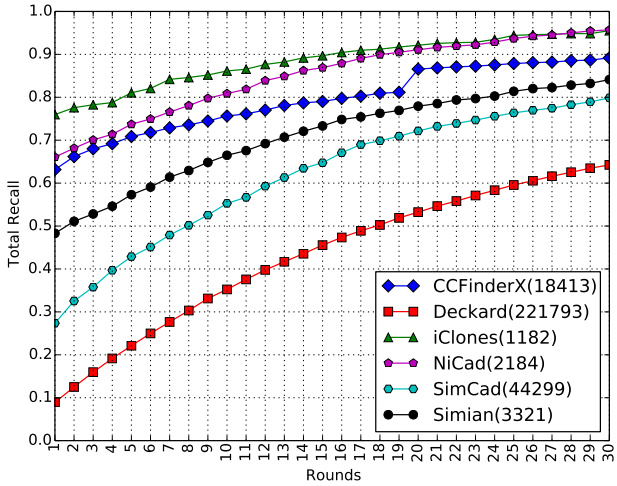


Figure 11.3: Preliminary Experiment - ArgoUML

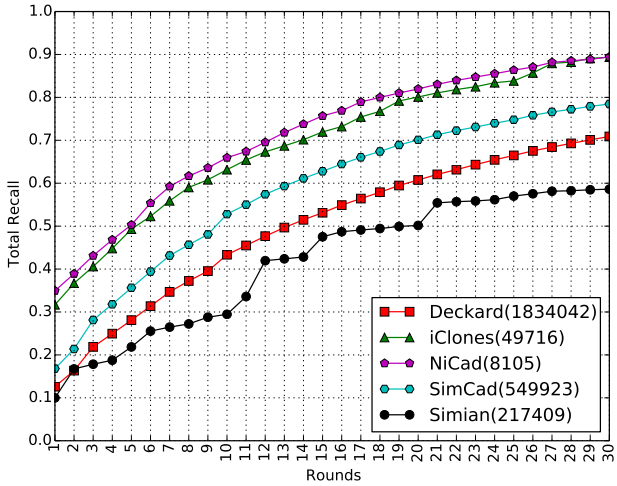


Figure 11.4: Preliminary Experiment - JDK1.7

data scenario. We were unable to obtain gold standards for the other tools. The required processing time and computer memory exceeded our available time and hardware rental budget. Simian is atypical in that it was scalable to big data within a reasonable execution time when a large amount of RAM was provided. However, Simian's detection capabilities are not as sophisticated as the other tools, for example, it only detected type 1 and type 2 clones.

Of the six selected tools only Simian, NiCad and Deckard were used successfully for this experiment. CCFinderX, iClones and SimCad were omitted due to compatibility issues with the dataset. These tools terminated with an error message if a parsing error was encountered instead of skipping the offending file. Since we used the largest subset size these tools could handle, the chance of a parsing error in a single subset is very high. This caused the tools to make very little progress in a round of the shuffling framework, as they used execution time but produced no clone reports for a large number of the subsets. The omitted tools are further discussed in Section 11.5.4. Table 11.2 summarizes the shuffling experiments performed.

11.5.1 Simian

Setup

Simian was chosen for this experiment as it was possible to obtain its gold standard for IJaDataset. This allowed us to compare our framework's performance with Simian against its native performance. For evaluation with the shuffling framework, a subset size of 50,000 files was chosen (58 subsets per round). Simian's fast execution allowed us to execute 30 rounds of the shuffling framework. Simian reports clones as clone classes, which was exploited when analyzing the results. We converted the clone classes into clone pairs to measure total recall.

Subset generation and round detection took approximately 8-12 and 4-10 hours per round, respectively, on Westgrid hardware. Based on our later experiments, we estimate that subset generation and construction would take approximately 1.25 hours, and detection approximately 1.25 hours per round, on our local machine with a solid state drive. The bottleneck on the Westgrid systems was the IO performance. Specifically, copying many small files from the dataset into the subsets was much slower on a traditional hard disk drive.

Table 11.2: Summary of the IJaDataset Clone Detection Experiments

Tool	Hardware	Subset Size (# files)	# Sets	# Rounds
Deckard	24GB	10K	289	10
NiCad	12GB	10K	289	20
Simian	12GB	50K	58	30

Analysis

Since Simian's gold standard is extremely large (300 billion clone pairs) total recall was estimated using the heuristic, which is shown in Figure 11.5. After 30 rounds of the framework, 70% of the cloned fragments in Simian's gold standard were detected. According to the heuristic study, Section 11.3.4, total recall should be less than the heuristic, but with faster growth. Specifically for Simian, the study showed that recovered recall quickly approached heuristic recall when total recall was within 70-90%. From Simian's heuristic recall trend for IJaDataset, we estimate it would reach 80% in approximately 10-20 additional rounds. We therefore conclude that Simian has achieved an acceptable recall for cloned fragments within 30 rounds. The heuristic recall suggests that within 10-20 additional rounds a transitive clone recovery technique could achieve an equivalent total recall of clone pairs.

While the heuristic is a worthy approximation of total recall, it is still desirable to directly measure total recall, which necessitated a reduction in Simian's output. Investigation into the characteristics of Simian's gold standard found that 99.99% of Simian's clones came from clone classes greater than 100 fragments in size. Manual investigation into these clone classes revealed that Simian suffered from what we term the "sliding effect". It reported some extremely large clone classes containing the same fragment(s) repeated numerous times with small offsets in line numbers. These clone classes generate an extreme number of self and overlapping clones and represent a significant threat to Simian's precision. We therefore reduced Simian's output size by trimming clone classes over a certain maximum size. Its gold standard was likewise trimmed. The remaining clone classes were converted to clone pairs to measure total recall. This post-processing of the framework's output for Simian was done solely to aid the evaluation of total recall on our hardware in a reasonable time-frame, and is not an expected post-processing step for users of the framework.

Figure 11.6 shows our framework's total recall with Simian for various maximum clone class sizes up to 100 fragments (limitation of our hardware). The legend of this figure specifies the maximum class size considered with the gold standard's size in parenthesis. Total recall was higher and increased faster for lower maximum clone class size. This suggests that the framework works best for specialized clone detection (i.e., focusing on detecting interesting/unique clones rather than all clones). This is due to larger classes requiring more rounds (on average) to be completely found as they contain more clone pairs that need to be shuffled together.

For the smaller class sizes a respectable total recall was achievable within 30 rounds (2: 52%, 5: 44%, 10: 40%). This total recall may be acceptable in cases where only a sample of the clones is required. For example, when building an inter-project clone corpus using many tools, 50% of a tools' native recall is likely sufficient for the corpus to benefit from the tool's unique detection characteristics. Consulting multiple tools may make up for individual tools' diminished recall.

While this total recall is low, in each case it is increases nearly linearly, with very little decay in slope. Additional rounds could bring these to an acceptable level. As can be seen, a 7-10% increase in total recall is gained per additional 10 rounds. A transitive recovery method could also help boost total recall achieved.

In our preliminary studies, we found that the framework performed the worst with Simian. Therefore we expect the other tools to achieve a higher total recall than Simian.

Figure 11.7 shows heuristic recall for the same trimmed output. As can be seen, the shuffling framework is finding the cloned fragments very fast, with 52-62% heuristic recall after only 30 rounds. Heuristic recall increases faster for larger maximum clone class size, meaning that the fragments in large clone classes are more easily found. This is expected as fragments in large clone classes have a higher chance of being shuffled into a partition with another fragment from the clone class. This suggests that a transitive recovery method may work especially well for the clones of large clone classes. This is particularly beneficial as it was for the clone pairs in larger classes that the framework had a slower increase in total recall (Figure 11.6).

11.5.2 NiCad

Setup

NiCad was included in this experiment for its ability to restrict clone detection to function clones. This is a beneficial functionality for clone detection within ultra large datasets as line level clones may be too numerous to process. Function clones are fewer, and may be more interesting as they occur at a higher level of software design. Function clone corpora built from ultra large inter-project datasets may be especially useful for mining new APIs.

Through experimentation, it was found that NiCad could consistently handle datasets of 10,000 files. It occasionally failed for larger input (e.g., 25K, 50K) due to hard coded limits in the sizes of its internal data structures. These internal limits appear to be intentional and designed to prevent users from beginning executions that are likely to fail or never complete on standard workstations. The internal data structure sizes can not be specified by the user without source code modification and recompilation. We left NiCad as-is for our goal is to scale the tools without modifications.

Based on these observations a subset size of 10,000 files was chosen for running the shuffling framework (289 subsets per round). As the framework achieved better total recall with NiCad than with Simian in the preliminary experiments and previous work [64], 20 rounds was deemed sufficient for demonstration of the framework. Subset generation and detection took 7-15 and 23-31 hours per round respectively on shared computing resources. Based on our later experiments, we estimate that generating and building the subsets would require 1 hour per round, and detection 17 hours per round, on a solid state drive.

Analysis

Creating a gold standard for NiCad was not possible, so we could not evaluate total recall. Internal data structure limits prevent NiCad from being executed on such a large input. Even if we modified these limits, NiCad would have required more RAM than we had available and likely months of execution time. We investigated using a deterministic partitioning technique (Section 11.1) to build NiCad's gold standard, but

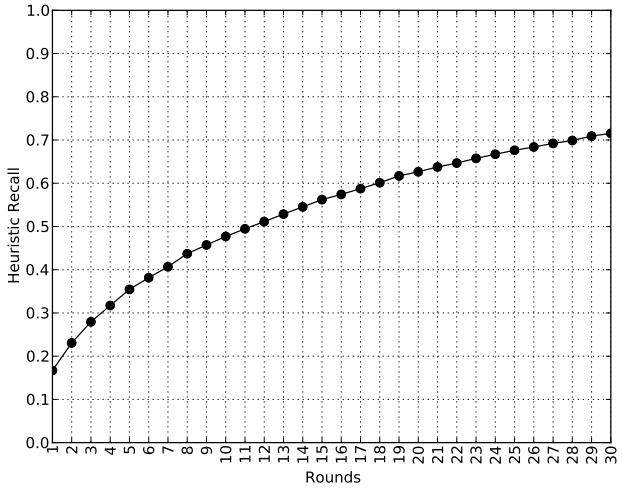


Figure 11.5: Simian Heuristic (Clone Fragment) Recall

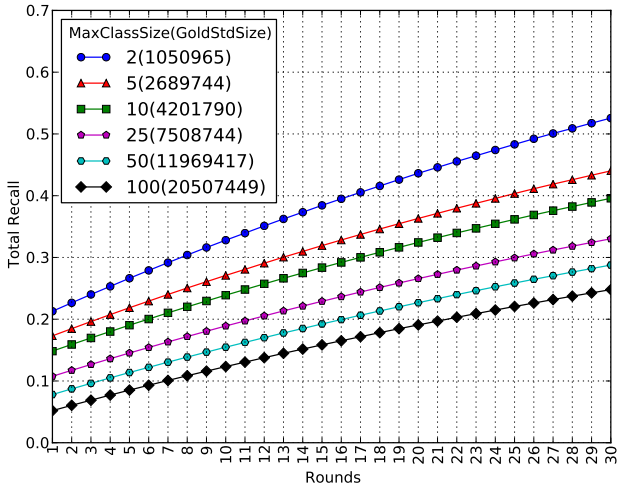


Figure 11.6: Simian Total Recall for Maximum Class Size Trimmed Output

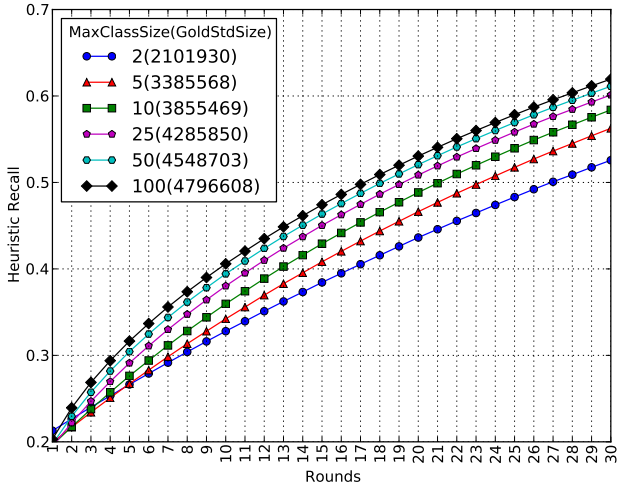


Figure 11.7: Simian Heuristic Recall for Maximum Class Size Trimmed Output

our estimates found that this would have required 2 months of execution time on our available hardware, even with four NiCad instances executing in parallel. Instead we investigated the growth of the cumulative number of unique clones and cloned fragments found after each round of the framework. This information is plotted in Figure 11.8. In total, 5.66 million unique clone pairs containing 875 thousand unique cloned code fragments were found.

The growth of unique detected clone pairs (Figure 11.8, diamond-line) is linear across the twenty rounds. This tells us that the framework has not detected a large ratio of NiCad's native recall. Linear growth per round tells us that the probability of undetected clones being shuffled together has remained constant over the rounds. Had a considerable ratio of NiCad's native recall been found, this probability should have also considerably decreased. Our preliminary study (Section 11.4) with small systems showed that the growth would appear logarithmic as the framework approaches a considerable ratio of the tool's native recall. Therefore, we require more rounds of the shuffling framework to achieve an acceptable ratio of NiCad's native recall.

In contrast, we do see logarithmic growth in the detection of unique cloned fragments (Figure 11.8, square-line). Per round, the number of new cloned fragments found is decreasing noticeably. It is becoming less probable that a clone (in NiCad's native recall), that contains an undetected cloned fragment, is randomly shuffled into a partition. This can only happen if a considerable ratio of NiCad's native cloned fragment recall is achieved per round. The growth has considerably declined after 20 rounds, suggesting that a considerable heuristic recall has been achieved.

These plots suggest that the framework is achieving a good heuristic recall with NiCad (the cloned fragments are being round quickly), but that the clone relationships between them (total recall) are still being detected. Applying a transitive clone recovery technique could recover some of the remaining clones without executing further rounds. As seen in the heuristic study (Section 11.3.4), clone recovery is very successful for NiCad. However, in that study, we applied transitivity naively to see the ideal results. To apply it in practice an efficient and accurate method for checking the validity of transitivity for type 3 clones would need to be designed and implemented.

11.5.3 Deckard

Setup

Experimentation found that Deckard worked for our approach with a subset size of 50,000 files, and could possibly work for larger subsets up to the entire dataset (untested). However, its execution time for large inputs was prohibitive (scaling limitation), so a subset size of 10,000 files was used to match NiCad (289 subsets per round). As Deckard has a lengthy execution time, the shuffling framework was executed over only 10 rounds. Detection was ran on Amazon EC2 and took approximately 5-7 days per round. While this execution time is very long, it is practical compared to Deckard's native execution time for the IJaDataset

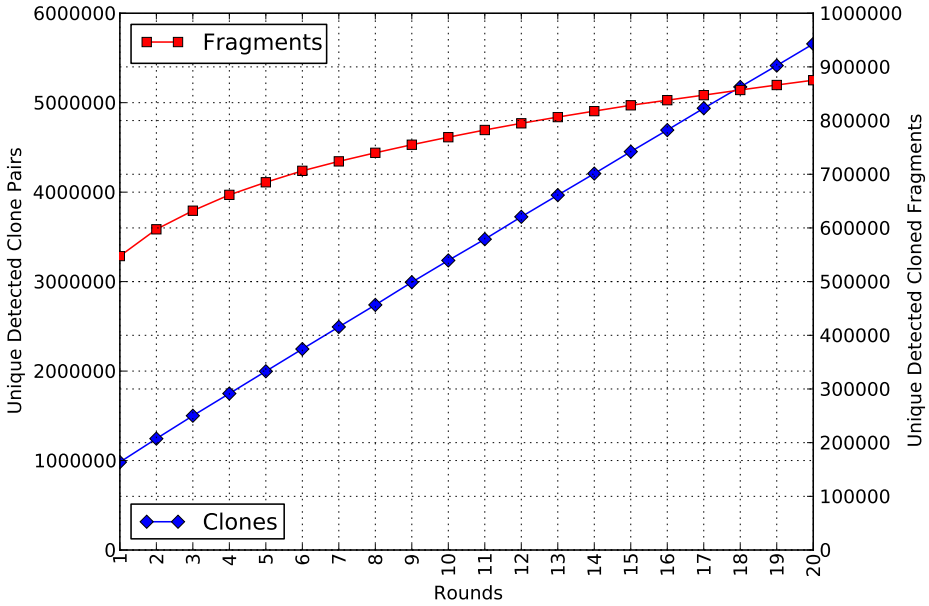


Figure 11.8: Growth of NiCad's Found Clones and Cloned Fragments

input.

Caveat

One disadvantage of Deckard is that it only supports up to Java 1.4 syntax. Its documentation specifies that it is able to skip unsupported syntax without error. In our experience, it found plenty of clones despite this limitation.

Analysis

Creating a gold standard for Deckard was not possible due to the computation time required, so we could not investigate total recall. Instead we investigated the number of unique clones and cloned fragments detected across the rounds as we did for NiCad.

Figure 11.9 shows the growth of the number of unique detected cloned fragments. As can be seen, the growth of detected cloned fragments follows roughly a logarithmic trend. The probability that the random partitioning shuffles a clone (in Deckard's recall) containing an undetected cloned fragment into the same partition decreases as heuristic recall increases. The considerable decrease in the number of new cloned fragments detected per round suggests that this probability is also considerably decreasing, and thus a considerable heuristic recall has been found. Unfortunately, we could not measure the detected clone pairs across the rounds due to the size of Deckard's output. We can infer from NiCad's and Simian's results it would likely be increasing linearly over these rounds.

In order to confirm our inference, we measured found clone pairs and fragments on a reduction of Deckard's output. We reduced the output sized by considering only reported clone classes with a maximum size of 10

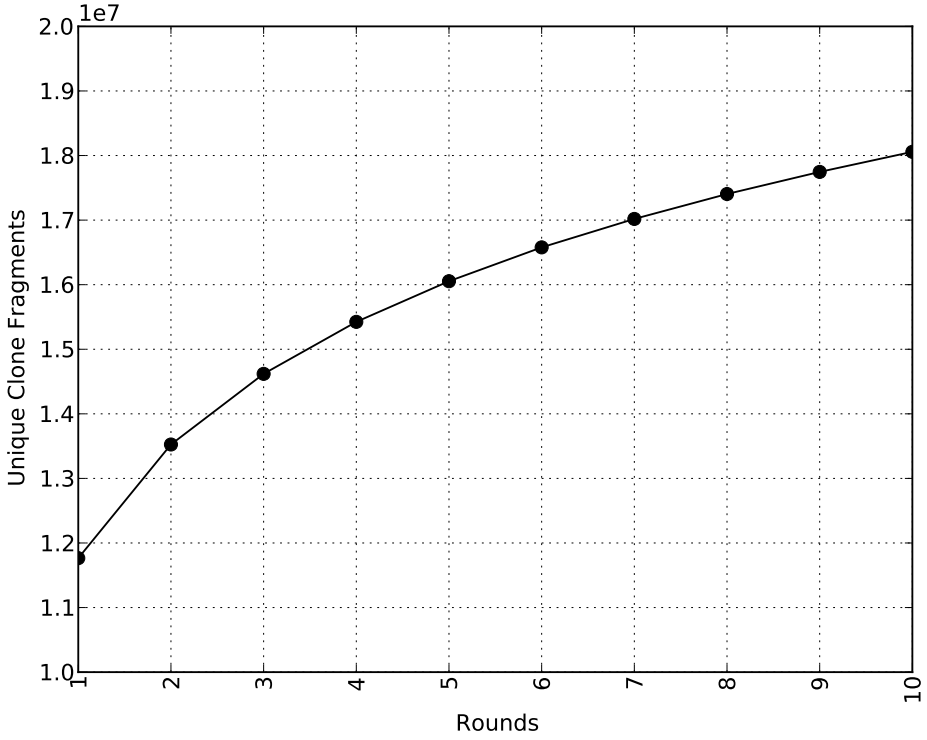


Figure 11.9: Growth of Deckard's Detected Cloned Fragments

fragments (limitation of our hardware). The growth of detected clones and fragments for this reduced output is shown in Figure 11.10. As expected we found very similar results to NiCad. The detected clones increase linearly, while the detected fragments shows logarithmic growth. This suggests that the cloned code fragments are being found before the clone pairs between them, and that a transitive clone recovery method would be successful in detecting additional clone pairs.

11.5.4 Other Tools - SimCad, iClones, CCFinderX

Our intention was to include SimCad, iClones and CCFinderX in the main experiment as they showed promise in the preliminary experiment. During evaluation of a sample from the dataset, these tools terminated without producing a clone report. SimCad and iClones reported encountering an invalid Unicode character. CCFinderX failed silently, but we believe this is due to the dataset containing Java code of a newer specification than CCFinderX can parse.

These problems do not indicate special scalability issues with these tools or with our framework. However, the framework can not make progress with tools that abandon detection when parsing errors are found, instead of trimming the offending file(s). Since we are executing these tools for partitions of IJaDataset near the limits of their scalability, there is a high chance that they encounter at least one parsing error. The tools fail upon the first parsing error detected, so it is not practical to compile a list of offending files in order to trim the dataset.

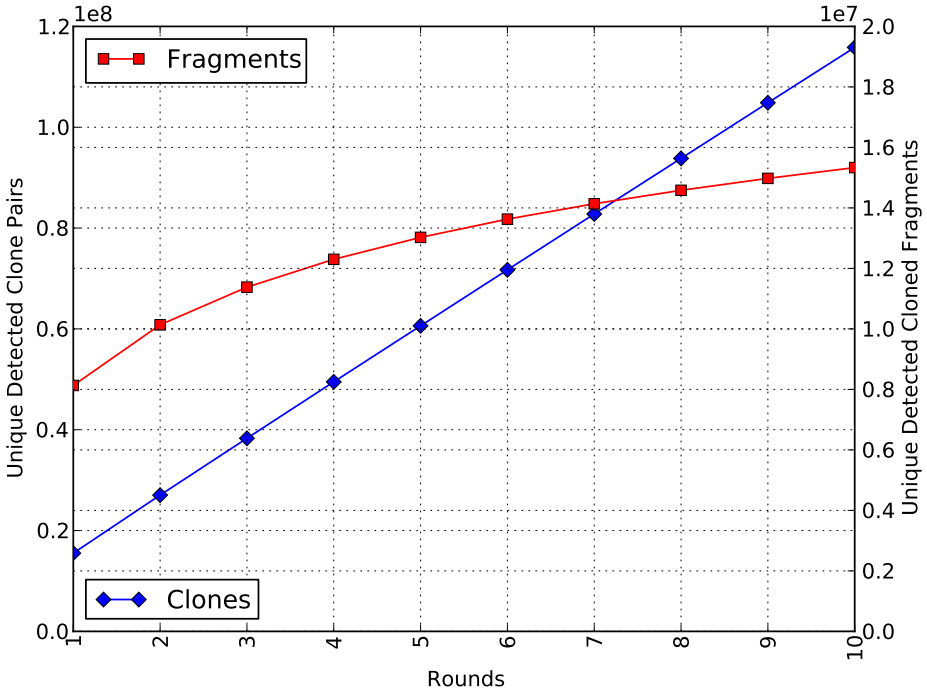


Figure 11.10: Deckard's Clone and Fragment Detection for Trimmed Output

Communication with the iClones developers revealed that this problem was fixed in a development branch, so we included iClones in our experiments in improving the Shuffling Framework (Section 11.7). SimCad was also corrected upon communication with the developer, but not in time to be used in this publication. We plan to revisit SimCad in future work. CCFinderX is no longer under active development so we do not anticipate improvements in its parsing or error handling.

11.5.5 Summary

From these experiments, we found that the clones found by the framework increased nearly linearly, with a slight decay in slope, across the rounds. This shows that additional rounds would continue to see a healthy increase in found cloned pairs, and thus an increased total recall. For Simian and considering only clone pairs originating from smaller clone classes (2-100 fragments) 25-52% total recall was achieved over 30 rounds, with a (decaying) continued increase of 7-10% per 10 rounds (Figure 11.6). Further rounds could bring total recall to an acceptable value.

However, the framework was able to find the clone fragments much faster. For each tool, found cloned fragments experienced logarithmic growth across the rounds. The decay in detection rate indicates that the probability of a clone containing an undetected clone fragment is randomly shuffled into a partition is decreasing noticeably. This indicates that the number of remaining undetected clone fragments is decreasing considerably. With Simian, 70% of the cloned fragments were found within 30 rounds (Figure 11.5).

These findings suggest that our framework finds most of the cloned fragments in few rounds, but may

require a large number of rounds to find all of the clone relationships between them. This suggests that a transitive-based clone recovery process could improve total recall achieved. This is supported by our heuristic study (Section 11.3.4), which showed that a strong heuristic (clone fragment) recall can be translated into a strong total (clone) recall by transitive recovery. Implementing an efficient and precise recovery process is therefore a priority for our future work.

From our experiment, we conclude that the shuffling framework is successful in scaling classical clone detection tools to ultra large datasets (**RQ#3**), but many rounds may be needed to achieve a high total recall. The framework is best suited for applications that accept partial clone detection tool recall as sufficient. For example, when building a comprehensive inter-project clone corpus (e.g., for IJaDataset) using a variety of both classical and scalable detection tools, 60-80% of a classical tool's native recall is likely sufficient to ensure the clone corpus benefits from its diverse strengths and detection characteristics.

The framework is very suitable for applications that only require knowledge of the cloned fragments within an ultra large dataset, and not the pairs. Given that we encountered scalability limits (memory and time) in processing the clone pairs found by this experiment, it is likely that studies on inter-project clone corpora of similar scale may need to be done on cloned fragments. Analyzing the clone pairs presents an additional big data challenge.

11.6 Shuffling Framework Performance Analysis

As seen in the IJaDataset experiment above, the shuffling framework is able to scale classical tools to large datasets. It is able to tackle various scalability issues, including: memory requirements, computation complexity, computation time, and internal tool limitations. However, we observed some inefficiencies in the original algorithm.

In the IJaDataset experiments with the core shuffling framework a suitable clone fragment recall was obtained (e.g., Figure 11.5). However, clone pair recall was much lower (Figure 11.6). To obtain a higher clone pair recall many more rounds would need to be executed, which would require considerable computation time. Alternatively, post-processing could be used to recover some of the missed clone relationships between the detected cloned fragments. Previously we showed that clone transitivity is effective at clone recovery (Section 11.3.4). However, transitivity is only certain for type 1 and type 2 clones. Type 3 clones recovered by transitivity would need to be verified before accepted. Our experiences with the IJaDataset experiment indicate that post-processing may be computation and memory intensive. We had considerable difficulties processing the detection results for statistical reporting. A novel and efficient approach and considerable computer resources are likely required to apply transitive clone recovery with both high recall and precision.

We decided that the best way to improve the performance of the shuffling framework was to improve the shuffling algorithm itself (**RQ#4**). Our goal was to decrease the number of subsets of the dataset a tool had to be executed for to obtain an acceptable total recall. Looking at the tools' clone pair detection performance

for IJaDataset (Figures 11.6, 11.8, 11.10) we notice a common trend. A large number of clones are detected in the first round, followed by a lesser but steady increase in subsequent rounds. We studied this behavior and found that the large increase in the first round is due to the successful detection of all the intra-file clones in the tool's gold standard. This occurs because the first round exposes the clone detector to every file in the dataset. The remaining rounds advance the detection of the inter-file clone pairs in a tool's gold standard. Fewer subsets would be required if the shuffling algorithm focused on the detection of the inter-file clones in rounds 2 through n.

We identified two major characteristics of the core shuffling framework that slow the rate of inter-file clone detection. First, the shuffling framework has no sense of history. There is nothing to stop it from repeatedly shuffling the same files into the same subsets. The clone detector will find any inter-file clones between a pair of files the first time they are shuffled together. Repeated shuffling of previously seen file pairs does not advance inter-file clone detection, but uses computation time. An improved shuffling framework should discourage repeated shuffling of the same file pairs.

Secondly, the shuffling framework does not consider file contents when it shuffles. Likely only a small ratio of the dataset contains inter-file clones. It is wasteful to shuffle dissimilar files together. The framework would require fewer subsets if it preferred to shuffle together files that contain enough similarity to likely contain a clone as judged by the classical tool. In the following section we explore incremental improvements to the shuffling algorithm that address these two limiting characteristics.

11.7 Improving the Shuffling Framework

We identified in the previous section that the shuffling framework's performance suffers due to the shuffling together of the same files repeatedly, and the shuffling together of dissimilar files unlikely to contain clones. Addressing these problems is non-trivial as optimal solutions are not practical for ultra large datasets due to the high complexities of the required algorithms. Even sub-optimal solutions can quickly increase the complexity and execution time of the shuffling algorithm. Our goal is to trade detection execution time (i.e., fewer subsets) for shuffling execution time (i.e., more valuable subsets). For this to provide a performance gain the shuffling algorithm needs to maintain a lower complexity and execution time than that of the clone detection tools. That is the cost of building subsets that provide a larger increase in total recall must be less than the cost of simply executing the tool for more subsets.

We investigated three new shuffling algorithms which incrementally address the issues of the core algorithm. These include: the unseen pair, the unseen similar pair, and the inverted index shuffling algorithms. We now term our original algorithm the blind partitioning shuffling algorithm, because it builds its subsets by blind random partitioning of the dataset.

The three new algorithms use two rounds of shuffling. In the first round, the framework completely partitions the dataset into disjoint subsets, and the tool is executed for each of these subsets. This is the

same as a round generated by the original core algorithm (Section 11.2). This first round exposes the tool to every file in the dataset, which ensures that the framework does not miss any of the intra-file clones the tool is able to find. In the second round, the framework pursues the inter-file clones the tool is able to detect. Since the intra-file clones should have been detected in the first round, it is no longer important to expose the tool to every file in the dataset in round 2. Therefore the new algorithms drop the partitioning strategy in round 2. Instead they execute the tool for a series of subsets that prioritize the exposure of the tool to new inter-file clone detection experiences. These subsets may not completely partition the dataset, they may overlap, and round 2 may contain any number of subsets. The subsets of round 2 are the same size as those of round 1, and non-determinism is still exploited in their selection. The difference between the algorithms in how they choose these subsets. Their goal is to require fewer subsets (fewer executions of the tool) than the blind partitioning algorithm to achieve some target total recall.

We evaluated these algorithms for three clone detection tools: NiCad, iClones and Simian. NiCad and Simian were used in our previous experiments. We now include iClones which had been fixed after our IJaDataset experiment (Section 11.5) had been conducted. The shuffling framework performed very well with iClones in the preliminary experiments (Section 11.4), so it was ideal to include it as a subject tool in this experiment. We decided to skip Deckard because while the shuffling framework was able to improve its scalability, its still required long execution times.

We evaluated the algorithms using random samples of IJaDataset. We choose sample sizes large enough to be more representative of an ultra large dataset than the systems used in the preliminary experiments (Section 11.4), while still small enough that we could obtain each tool's gold standard for measuring total recall. For NiCad and Simian we used a dataset of 50,000 files randomly selected from IJaDataset. For iClones we randomly selected 10,000 IJaDataset files. For the 50,000 file sample a subset size of 250 files was used and 50 file subsets were used for the 10,000 file dataset. This is the same ratio between subset size and dataset size as used in the IJaDataset experiment (Section 11.5) for NiCad and Deckard. This way the evaluation of the algorithms approximates their usage with IJaDataset, or another ultra large dataset.

Performance was measured as total recall, the ratio of the clone pairs in a tool's gold standard the shuffling algorithm was able to find (Section 11.3, Eq. 11.1). In order to save time we simulated the execution of the tools. We assumed that for a particular subset, the tool would output the clones from its gold standard that are within or between files in the subset. For these deterministic code clone detection algorithms, this is a reasonable assumption.

By simulating the clone detection we are able to measure the framework's performance more accurately. Ideally a clone detection tool reports the same clones between a pair of files regardless of number and particular files also included in the input. In practice, the clone detector may not report clones consistently. It may report the same clones but with slightly different start/end lines, it may miss some clones or find additional clones. Its precision (the number of false positives reported) may also vary. This may be caused by bugs in the clone detector. By simulating the detection using the clone detector's gold standard, we avoid

these issues.

The total recall performance of these four algorithms are shown in Figures 11.11, 11.12, and 11.13. Their subset generation time is shown in Figures 11.14 and 11.15. The generation time includes only the time required to generate the list of files to be included in each subset. It does not include the time needed to copy the files into a temporary directory and execute the tool.

In the following subsections, the algorithms are outlined and their performance discussed. The algorithms are presented in the order in which they were created as to emphasize our design process and decisions.

11.7.1 Blind Partitioning Shuffling Algorithm

The blind partitioning shuffling algorithm is the original shuffling algorithm as presented in Section 11.2. It is the cheapest shuffling algorithm in terms of subset generation processing time and complexity. Its performance is the base line against which the other algorithms are compared. For both the 10,000 and 50,000 file datasets the blind shuffling algorithm partitioned the dataset into 200 subsets per round for 10 rounds. Remember that for each round this algorithm partitions the dataset into mutually exclusive subsets that together span the entire dataset. The result of its application for NiCad, Simian and iClones can be seen in Figures 11.11,11.12 and 11.13, respectively. Since the new algorithms do not use the same number of rounds, the total recall is plotted after each subset. The tick marks of the x-axis correspond to the rounds of the blind shuffling algorithm. The first round (in which all algorithms use the blind partitioning strategy) is labeled.

This algorithm achieves a large total recall increase for all tools within the first round. Since the first round exposes the clone detector to every file in the dataset, all the intra-file clones in the gold standard should be detected in this round. In the remaining rounds we see linear growth in total recall as the inter-file clones in the gold standard are located by chance due to blind random file shuffling. Linear growth was expected as the algorithm applies the same random dataset partitioning in each round. This linear growth should experience a decay in its slope as more clones are detected and it becomes increasingly less likely the random partitioning will shuffle the files containing the undetected clones into the same subset.

The increase in total recall due to inter-file clone detection in rounds 2 through 10 is much smaller than the increase due to inter and intra-file clone detection in round 1. Since the growth is linear across rounds 2 through 10, we expect roughly the same number of inter-file clones were detected in round 1. Therefore the larger increase in total recall in round 1 must be dominated by the detection of the intra-file clones.

11.7.2 Unseen Pairs Shuffling Algorithm

The first problem we identified with the blind shuffling algorithm is that it does not discourage the repeated shuffling together of the same files. Our efficient solution is to fill the subsets with randomly selecting pairs of files from the dataset that have not been shuffled into the same subset previously. This solution guarantees that each subset includes $n/2$ new inter-clone detection experiences (i.e., unseen file pairs), where n is the

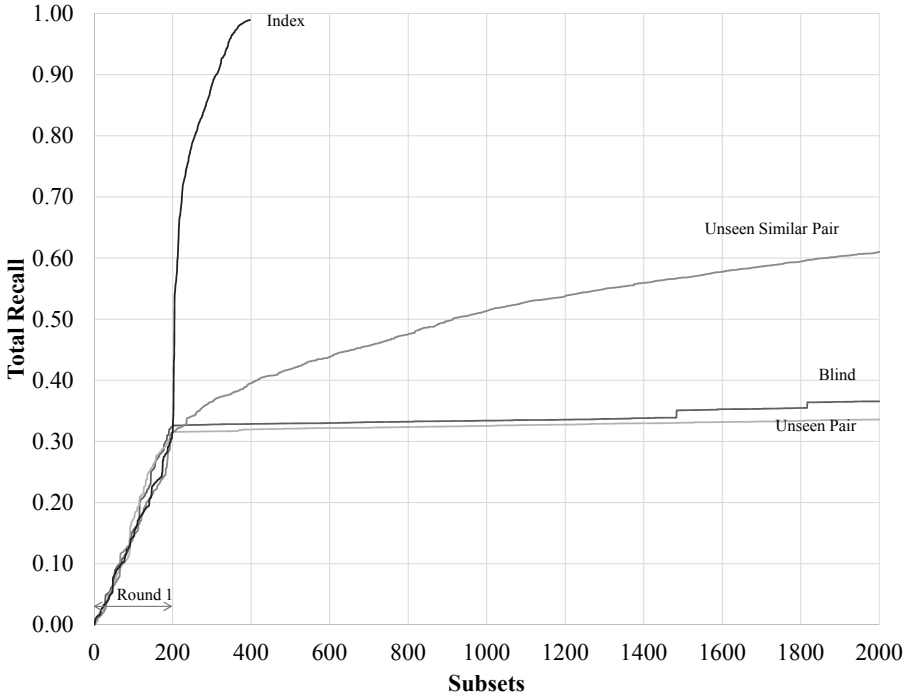


Figure 11.11: Shuffling Algorithm Performance Comparison: NiCad, 50,000 File Dataset, 250 File Subsets

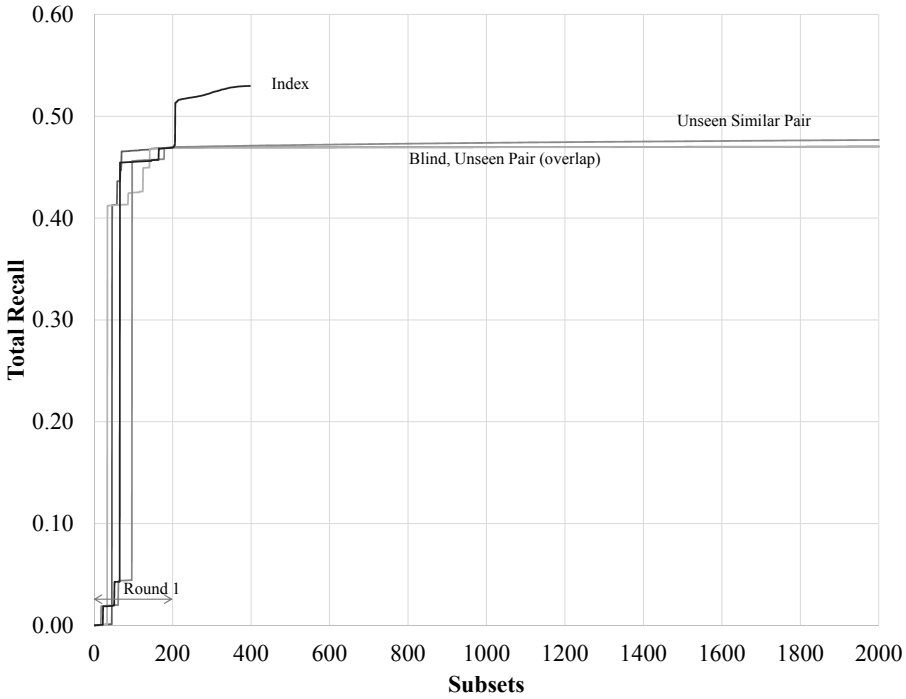


Figure 11.12: Shuffling Algorithm Performance Comparison: Simian, 50,000 File Dataset, 250 File Subsets

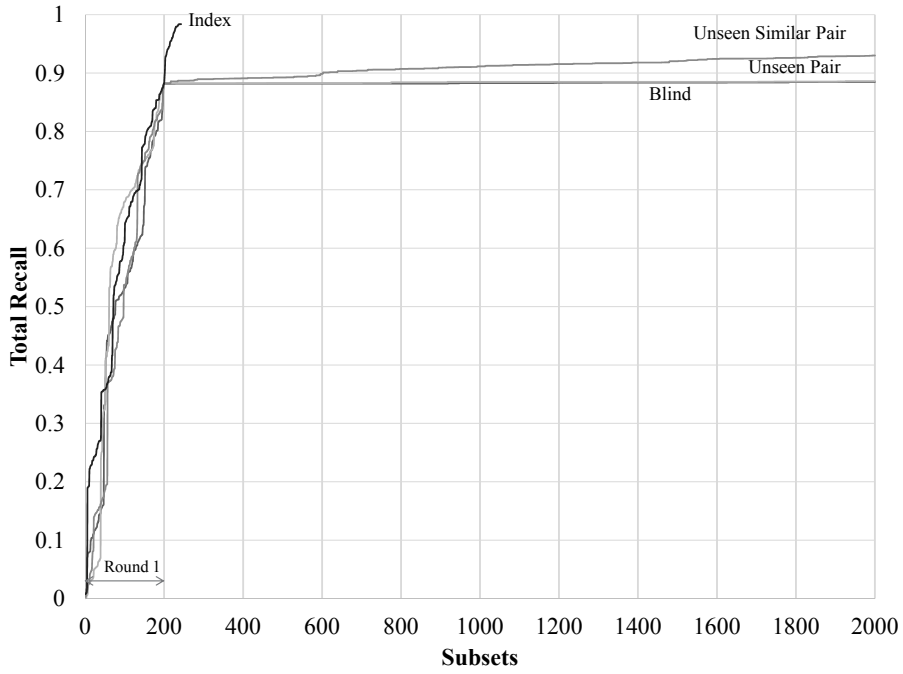


Figure 11.13: Shuffling Algorithm Performance Comparison: iClones, 10,000 File Dataset, 50 File Subsets

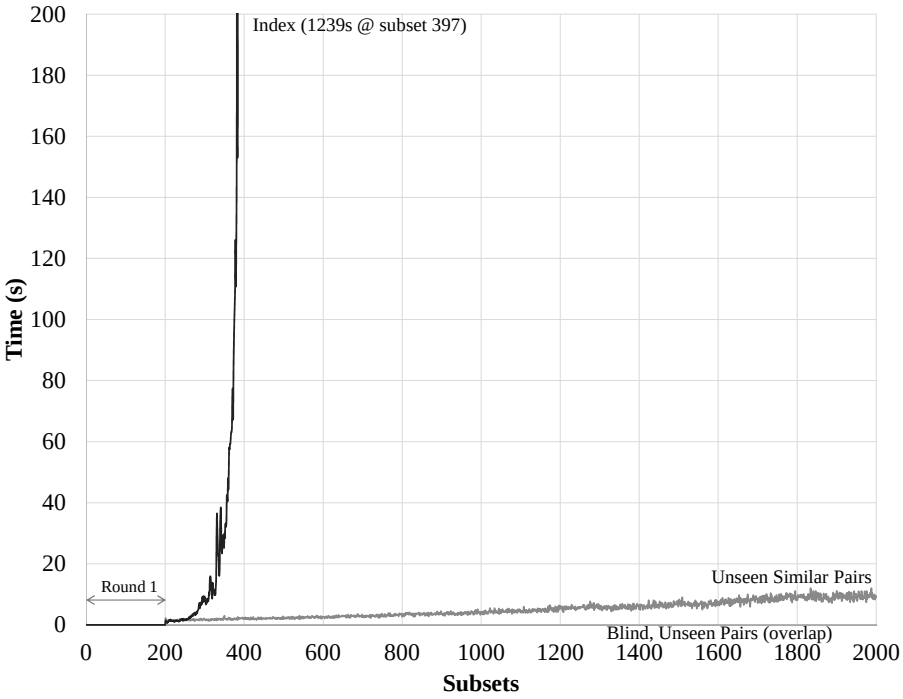


Figure 11.14: Shuffling Algorithm Computational Comparison: Subsets Generation Time (ms), 50,000 File Dataset (NiCad/Simian)

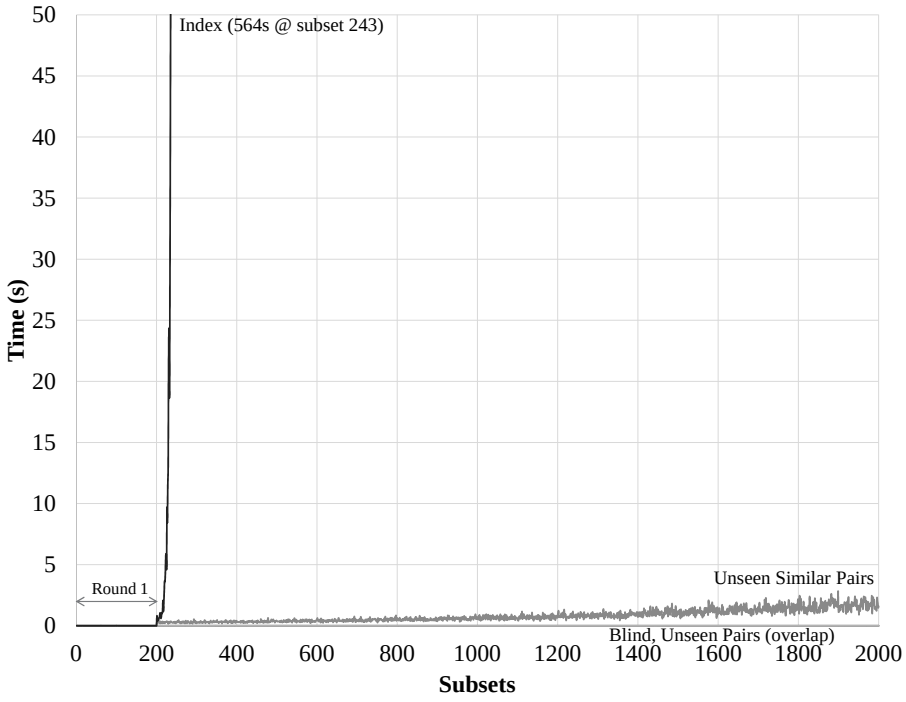


Figure 11.15: Shuffling Algorithm Computational Comparison: Subsets Generation Time (ms), 10,000 File Dataset (iClones)

number of files in the subset. This is the same number guaranteed by a deterministic approach (Section 11.1). However, in this non-deterministic case, it is likely that the files in a subset form many additional unseen pairs other than those specifically chosen. The worst case of only $n/2$ unseen pairs in a subset should only occur once most of the file pairs in the dataset have been seen in earlier subsets.

The first round of this algorithm uses blind shuffling to partition the dataset into subsets. This is needed as the unseen pair strategy does not guarantee nor encourage the detection of all intra-file clones in the dataset. By the end of the first round, every file has been seen by the clone detector. This first round does not detract from the unseen pair strategy as all subsets of the first round are made up completely of unseen pairs.

For round 2, the algorithm fills a user-specified number of subsets with unseen file pairs. Specifically, for each subset the algorithm performs the following steps:

1. Two files are randomly chosen from the dataset.
2. Efficiently checks if the pair has been seen in a previous subset.
3. If not previously seen, the files are added to the current subset.
4. If the subset is not full the algorithm repeats from step 1. Otherwise, the subset is complete.
5. The specification (file list) of the subset is saved.

6. The algorithm repeats from step 1 for some number of subsets.

This subset filling technique is efficient until most of the pairs have been seen, at which time the algorithm may cycle extensively until an unseen pair is found. For this reason, the algorithm is parameterized with a stopping condition: the number of times to cycle before giving up on being able to fill the subset. However, for an ultra-large dataset it is unlikely that this will occur for a practical number of subsets.

This algorithm needs to be able to efficiently check if two files have been seen together in a previous subset, step (2). This is accomplished by assigning each file in the dataset an id, and tracking the contents of each subset using a bit vector. Vector$[i](id)$ is 1 if the file with the specified id is in the ith subset. Shuffling an ultra large dataset like IJaDataset, which contains approximately 3 million files, requires only 0.35MB of memory per subset. The algorithm can determine if a file pair is unseen in worst case O(b) time, where b is the number of previously generated subsets. Examining a bit vector is very fast, so this linear search is acceptable. There are ways to structure the seen pair data to make O(1) possible, but it requires too much memory for the data structure to fit in RAM. We use this linear approach to avoid disk access times.

The unseen pair shuffling algorithm's total recall for the three tools is shown in Figures 11.11,11.12 and 11.13, and its subset generation time is shown in Figures 11.14 and 11.15. Remember that the first 200 subsets (round 1) are built using the blind partitioning strategy to ensure intra-file clone detection. This algorithm performed a little worse than the blind shuffling algorithm for the NiCad experiment, equal in the Simian experiment, and a little better in the iClones experiment. Its generation time is also essentially equivalent to that of blind shuffling.

Overall, the unseen pair shuffling algorithm does not perform any better or worse than blind shuffling. The reason for this is quite simple: the number of pairs in these datasets is so numerous that the blind shuffling technique is not shuffling the same files together repeatedly as frequently as we feared. However, the unseen pair strategy would be required if the number of potential file pairs was reduced. We decided to extend this algorithm to consider the second problem with blind shuffling: we should only shuffle together files that are similar. As this will likely reduce the number of potential file pairs considerably, we maintain the unseen pair selection strategy in the next algorithms.

11.7.3 Unseen Similar Pairs Shuffling Algorithm

The unseen similar pairs algorithm extends the unseen pairs algorithm to also address the second problem with the blind shuffling algorithm: it does not consider the contents of the files it shuffles together. Similar files are more likely to contain clones, so total recall obtained per subset could be achieved by prioritizing the shuffling together of similar files. Specifically, our goal was to parametrize the unseen pairs algorithm with a file similarity heuristic. Under this scheme, unseen file pairs are added to a subset only if the heuristic decides the files are similar enough to possibly contain a clone as judged by the classical clone detection tool.

The challenge was designing a suitable heuristic that can make a smart decision without significantly adding to the shuffling algorithm's complexity and execution time. Most clone detectors report a clone

between two files if they share a sequence of similar source code lines of some minimum size. Therefore, if two files contain this minimum number of similar lines, than it is possible the tool will find a clone between them. However, searching for similar line sequences between two files has polynomial complexity, which is also the complexity of most clone detectors. We needed a linear heuristic to ensure the improved shuffling algorithm was a performance gain.

Our heuristic accepts a file pair if the files share a minimum number of similar source lines. To do this in linear time we dropped the requirement that the similar lines need to be sequential. The disadvantage of this speedup is that the heuristic will accept file pairs with similar lines too sparsely distributed to be a clone. As source line (string) comparison is costly, we pre-computed hashcode (integer) values for each source line in the dataset. The number of shared lines between two files can then be calculated by measuring the size of the intersection of the hash-codes they contain.

To improve the heuristic's clone presence detection and accuracy we normalized and filtered the source code during the hash coding process. First, the source code was pretty-printed to normalize formatting. Identifiers were normalized (blind renaming) such that two source lines which differ only by identifier names will hash to the same value. Inconsequential, but common, source lines (e.g., '}') were removed to reduce the heuristic's false positive rate. Comments were also removed before hashing as most clone definitions and detection tools ignore them.

The heuristic is parametrized with the minimum clone size (in identical lines) of the target tool's configuration. For example, if the tool is set to report clones 10 lines or larger with a minimum of 70% similar lines, then the heuristic should be configured for 7 identical lines. Specifically, this is the minimum number of identical lines in a type 3 clone as judged by this tool. The heuristic will then accept file pairs that could contain a clone as judged by the tool, and reject those that do not contain sufficient similar lines for the tool to report a clone.

Mismatch between how the framework and the tool count source lines may cause some file pairs containing a clone detectable by the tool to be rejected. Mismatch may occur due to differences in how the framework and the tool normalize and filter input source code. Also, when tools measure minimum clone size by tokens, then minimum lines needs to be estimated. Rejected file pairs that contain a clone as judged by the tool will lead to additional false negatives. This should only occur for small clones that are near the minimum clone size, and whose files contain no other similar lines. The latter because the heuristic does not consider the position of the lines when measuring the number of lines shared between two files. This is acceptable as smaller clones are more likely to be spurious or uninteresting. Compensating for mismatch by setting the minimum clone size lower than that of the tool is a bad idea as it will cause the heuristic to accept more file pairs that do not contain clones, which lowers the effectiveness of the heuristic. With the heuristic we trade some of the tool's recall for smaller clones in exchange for fewer subsets to meet a target portion of the tool's native recall (total recall).

For the evaluation of this algorithm, we parametrized the heuristic with a minimum similar line threshold

of 5 lines. By default Simian detects type 1 and 2 clones with a minimum of 6 lines, iClones a minimum of 100 tokens (~5-10 lines) with gaps, and NiCad a minimum of 10 lines (30% of which may be gap lines). The 5 line threshold should be conservative enough to not skip too many file pairs that contain clones as judged by these tools. The algorithm's total recall performance is shown in Figures 11.11, 11.12 and 11.13, and its subset generation time is shown in Figures 11.14 and 11.15. Again, its first 200 subsets are its first round of blind shuffling to allow full intra-file clone detection.

For our 50,000 file dataset, subset generation time begins at approximately 1.5 seconds, and increases linearly as more subsets are generated. The generation time increases as the number of remaining unseen and similar file pairs in the dataset decreases. It takes longer to randomly locate an eligible file pair as they become more rare. This is not a defect as the rarer the eligible pairs become, the higher the total recall the shuffling algorithm has obtained. The number of subsets to generate is therefore a balance between the total recall goal and available subset generation time.

For all three tools, the unseen similar pair algorithm achieves a higher total recall then both the blind partitioning and unseen pairs shuffling algorithm within the same number of subsets. This increase is most pronounced with NiCad and iClones, while only a small increase is achieved with Simian. The algorithm's total recall for Simian may be higher if a more conservative similar line threshold were used.

Smaller gains with Simian might indicate the tool has worse precision. This shuffling algorithm only encourages the shuffling together of file pairs that may contain a true positive clone. The measurement of total recall does not consider if the clones in the tool's gold standard are true or false positives. Precision deficiencies of the tool would manifest in this evaluation as poorer total recall measurement. However, we do not have sufficient information about Simian's precision to conclude this.

11.7.4 Inverted Index Algorithm

The unseen similar pairs algorithm successfully increases the performance of the shuffling framework. However, we believed that considerably better performance could be achieved if the heuristic was sensitive to the locations of the shared source code between the files. The heuristic needed to be able to detect if two files had similar lines that were also closely located (e.g., subsequent). We were able to achieve this without increasing the complexity of the heuristic by computing n-grams across the hashed source lines of the dataset that were used with the previous algorithm.

The n-grams were calculated by summing each n subsequent hashed source lines using a sliding window. For example, a file with the hashed source lines A, B, C, D, E has as a 3-gram representation of $(A + B + C), (B + C + D), (C + D + E)$. The heuristic then approves a file pair if they have a minimum number of similar n-grams between them. For our evaluation of this algorithm we used a 3-gram representation. We pre-computed this in linear time in a single pass across the hashed version of the dataset.

The heuristic was parametrized to accept file pairs with at least three shared 3-grams. In effect, the heuristic considers two files to contain a clone if they share at least three incidents of three similar and

subsequent original source lines, which may overlap. File pairs approved by the heuristic therefore have at minimum between 5 (totally subsequent) and 9 (3 incidents of 3 subsequent) similar source lines.

During our initial investigation of this algorithm we found it had a very lengthy subset generation time. It was spending a large amount of time randomly selecting file pairs from the dataset that did not satisfy the heuristic. We minimized this problem by selecting the file pairs from an inverted file index built for the n-grams. The index maps each n-gram value to the files that contain at least one incidence of that n-gram. By randomly selecting file pairs from the index we are guaranteed they share at least one n-gram. This considerably reduces the selection space and allows the subsets to be built faster. The inverted index was represented by a hash map, and built in linear time by a single pass across the n-grams.

The inverted index can still be too large of a search space. Some n-gram appear very frequently within a dataset. To counter-act this, the index is trimmed of the n-grams that appear in over a maximum number of files. Less common n-grams are more likely to denote a clone rather than common structural/stylistic code (e.g., series of declaration statements at the beginning of a function). For our evaluation, we set the n-gram appearance threshold for our inverted index to 1000 files.

The evaluation of this algorithm with NiCad, iClones and Simian for 3-grams, a minimum of three shared 3-grams between file pairs, and an index n-gram appearance threshold of 1000 files is shown in Figures 11.11, 11.12 and 11.13. The subset generation times are shown in Figures 11.14 and 11.15.

Subset generation time increased exponentially as more subsets were created. We stopped the generation after 397 subsets (the first 200 of which were the round 1 blind shuffling subsets) because the subset generation time had increased by two orders of magnitude. The high generation cost means that file pairs that both satisfy the n-gram similarity heuristic and remain unseen have become rare in the search space (inverted index). It is taking a long time for the random selection process to find a suitable pair. The fact that the algorithm is reaching a high generation cost so quickly means that it is reaching its maximum total recall potential in fewer subsets than the other algorithms.

Not only does this algorithm exhaust its search space in fewer subsets, these subsets provide much higher increases in total recall. With NiCad and iClones, this algorithm achieved a very high total recall using far fewer subsets than the other algorithms. With these tools, and given a sufficient number of subsets, nearly 100% of the tool's native recall was achieved. Considerable gains were also seen with Simian compared to the other shuffling algorithms, but end total recall was much lower. The heuristic settings may not have been conservative enough for the types of clones Simian detects. Total recall would also be low if Simian has low precision. This shuffling algorithm avoids shuffling together files that are not similar as judged by the file similarity heuristic. A inter-file false positive in Simian's gold standard may never be shuffled together. However, we can not conclude this as we do not know Simian's precision performance for large inputs.

Since subset generation time increases so rapidly, we decided to investigate how quickly it increased with respect to total recall achieved. We plot this for NiCad and Simian (50,000 file dataset) in Figure 11.16, and for iClones (10,000 file dataset) in Figure 11.17. With NiCad, subset generation time had only increased by

a single order of magnitude (1 to 10 seconds) by the time a 90% total recall was achieved. This is up from 32%, the total recall after blind partitioning in round 1. To reach a near 100% total recall, another order of magnitude increase in subset generation time was required. The exponentially increasing cost of subset generation only becomes severe after most of the total recall has been achieved, which is very acceptable.

We see a similar trend for iClones. Total recall increases from 88% to 98% within the first order of magnitude increase in subset generation cost. We then see very little gains in total recall for the next order of magnitude increase. The primary difference from NiCad is that the framework had a high total recall with iClones after the first round (blind partitioning). This was because we had to use a dataset 5x smaller for this experiment with iClones, due to iClones memory requirements. A smaller dataset will have a larger intra-file to inter-file clone ration. The average incidence of intra-file clones for samples of IJaDataset will remain constant no matter the sample size. However, for larger samples the average incidence of inter-file clones will be higher because the number of file pairs potentially containing clones increases polynomially with sample size.

11.7.5 Choosing an Algorithm

Of the four algorithms proposed, the inverted index shuffling algorithm achieves the highest total recall in the fewest subsets. Its subset generation time is much longer due to its smaller search space, but this is an advantage since it is converging to a high total recall quickly. This equates to far fewer executions of the clone detection tools, which have higher complexities and longer execution times than subset generation. Using our simulated experiments using smaller datasets, we have demonstrated success for **RQ#4**: by observing the performance of the shuffling framework and incrementally responding to our observations, we were able to significantly improve its performance.

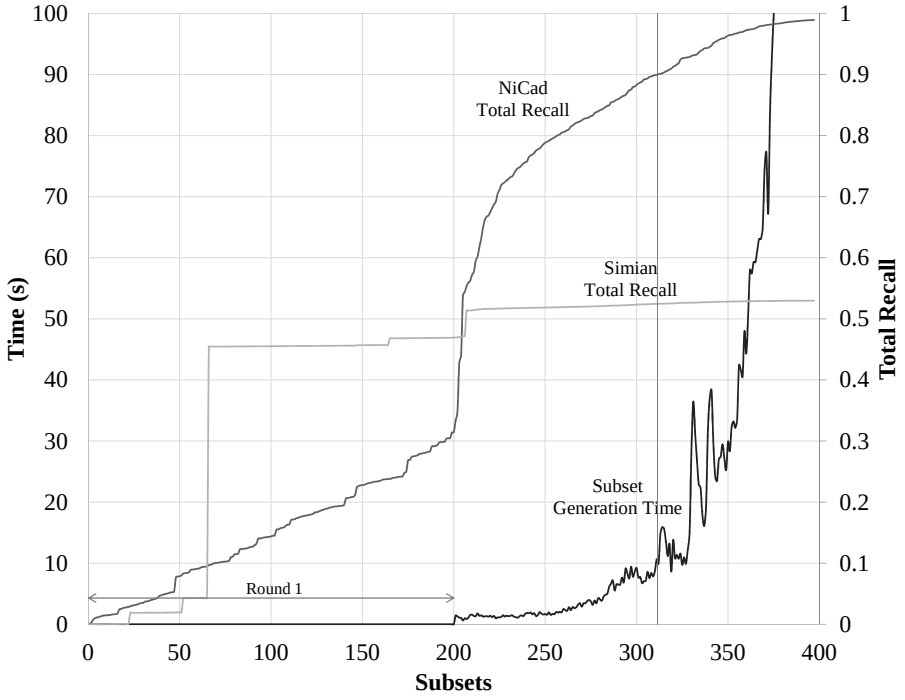


Figure 11.16: Inverted Index Algorithm - Subset Generation Time vs. Total Recall (NiCad/Simian, 50,000 File Dataset)

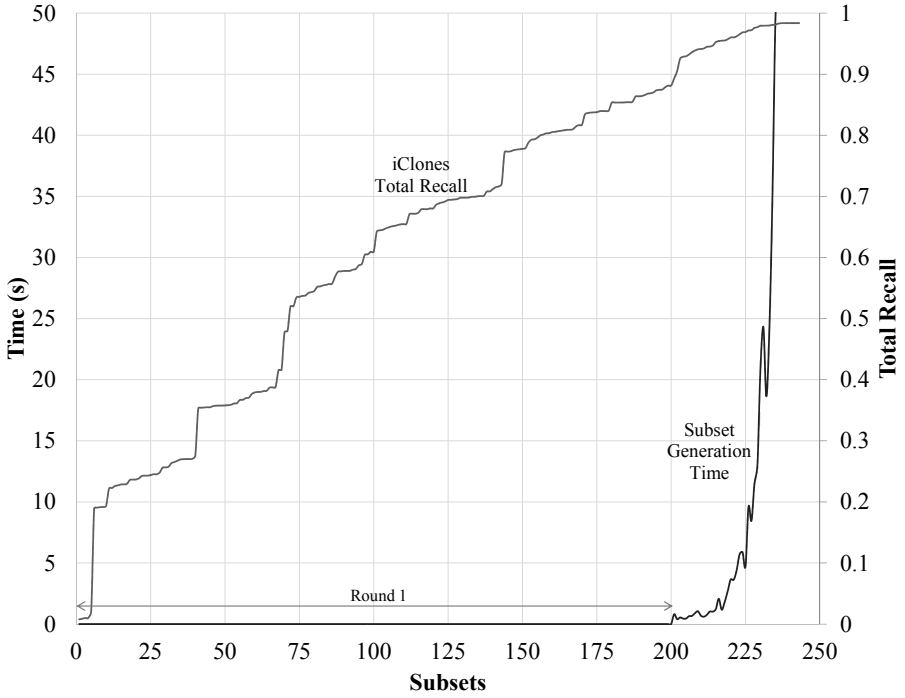


Figure 11.17: Inverted Index Algorithm - Subset Generation Time vs. Total Recall (iClones, 10,000 File Dataset)

11.8 The Improved Shuffling Framework

In this section we summarize the improved shuffling framework as developed in Section 11.7. The improved shuffling framework executes a classic tool for two rounds of subsets of some "big data" input source code dataset. The first round uses the blind partitioning shuffling algorithm, while the second round uses the inverted index shuffling algorithm. Consider the tool's hypothetical gold standard for the ultra large dataset. Round 1 will enable the tool to detect all of the intra-file clones in the gold standard (in addition to some inter-file). Round 2 aims to enable the tool to detect a large ratio of the remaining inter-file clones in the gold standard in as few subsets as possible. The steps of the improved framework are summarized in Figure 11.18.

We demonstrated in Section 11.7 that this method can achieve up to 98% of a classical tool's native recall given a sufficient number of subsets. However, subset generation became very expensive when 90% of the tool's native recall was achieved. The improved shuffling framework must generate the subsets serially as it needs to track which pairs of files have been seen in a previous subset, necessitating a definite subset order. However, the clone detection tool can be executed as soon as the first subset has been generated.

The improved framework guarantees that each subset contains at least $m/2$ unseen file pairs, where m is the size of the subsets. However, until the majority of the search space has been investigated, the subsets should contain far more unseen pairs. The similarity heuristic ensures that the guaranteed unseen file pairs also contain enough similarity to possibly contain a clone. This is done by selecting an n-gram size, n, and minimum shared n-grams heuristic, s, with respect to the tool's minimum clone size in identical source lines. If two files contain exactly s n-grams in common, then they share between $n + s$ (sequential) and $n * s$ (s occurrences of n sequential) source lines. So n and s should be picked with respect to the tool's minimum clone size. We had good results using 3-grams. A trim threshold for the inverted index, t, must also be selected. We had good results trimming the index of any n-gram that appears in over 1000 files.

11.8.1 Comparison with Deterministic Method

The improved shuffling framework is a non-deterministic method for scaling classical tools. The most similar previous work to our framework is the deterministic method described in Section 11.1. The deterministic method scales tools by: (1) partitioning the dataset into partitions half the size of the tool's maximum input, and (2) executing the tool for each unique pair of partitions. The deterministic method achieves 100% of a tool's native recall after $\frac{x(x-1)}{2}$ subsets. x is the number of partitions and $x = \frac{r}{0.5m}$, where r is the size of the dataset and m is the tool's maximum input size.

Consider if we split the deterministic method's subsets into two rounds, as we did with our non-deterministic shuffling framework. Both methods have $\frac{r}{m}$ (or $\frac{x}{2}$) subsets in their first round, and each of these subsets have $\frac{m(m-1)}{2}$ unseen file pairs. In the second round, the deterministic method has exactly $\frac{m}{2}$ unseen file pairs per subset. Each of the deterministic method's partitions have been seen in round 1, so the only unseen pairs are those between the joined partitions. Our shuffling framework guarantees its subsets in round 2 contain

1. Preparation
   (a) The maximum input size measured in source files, m, a clone detection tool can reliably handle on standard hardware is measured.
   (b) A hashed version of the dataset is generated. Each source line is replaced by a integer hashcode. The dataset is normalized (pretty-printed, identifier renaming) and filtered (of common structural lines) before hashing.
   (c) The hashed version is replaced by sliding n-grams, for some value n.
   (d) An inverted index is built, which maps each n-gram value to the files that contain at least one incidence of that n-gram.
   (e) The t most popular n-grams are removed from the inverted index, in order to reduce the search space.
2. Subset Generation: Round 1 - Blind Partitioning
   (a) The source files of the dataset are randomly partitioned into non-overlapping subsets of size m.
   (b) The specification (file list) of each subset is stored.
3. Subset Generation: Round 2 - Inverted Index
   (a) A subset of size m is constructed by randomly selecting pairs of files from the inverted index that share at least one n-gram. A pair is added to the subset if (1) the files have not previously been in the same subset, and (2) the files share at least s n-grams.
   (b) Once the subset is full, its specification (file list) is stored.
   (c) Subsets are generated until either (1) some user-specified maximum has been reached, (2) some maximum number of randomly selected file pairs have been rejected in a row, or (3) the user interrupts the process.
4. Clone Detection
   (a) A subset's specification is retrieved and the subset is constructed.
   (b) The tool is executed for the subset.
   (c) The tool's clone detection report is merged into a clone repository that efficiently removes duplicates (e.g., hash set, indexed database table).
   (d) Clone detection on the subsets may be done serially, in parallel or distributed. Clone detection can begin as soon as the first subset has been generated. A subset can be analyzed as soon as its specification has been completed.

Figure 11.18: Improved Shuffling Framework Procedure (Summary)

at least $\frac{m}{2}$ unseen file pairs, but should contain many more until the majority of the search space has been explored. The shuffling framework's similarity heuristic means that its subsets will contain more clones on average than the deterministic method.

Our shuffling framework makes progress faster (i.e., using less subsets) then the deterministic method. However, the deterministic method is better for some use cases. As seen in Section 11.7, the improved shuffling framework hits a point of diminishing returns before 100% total recall is achieved. The cost of generating the subsets becomes too costly. So the shuffling framework is appropriate for cases where some sacrifice in a tool's native recall is permissible. The deterministic method is needed when 100% native recall is required. Note that no clone detector has perfect recall, so 100% native recall does not mean perfect output.

The subsets of the shuffling framework need to be generated sequentially. There is a limit to how many computers the execution of the clone detection tool for the subsets can be distributed across before serial subset generation becomes a bottleneck. In contrast, the deterministic method's has marginal subset generation computation cost, and could be distributed up to one computer per pair of partitions. So if a large cluster of computers is available, the deterministic method may be the better option.

In summary, the shuffling framework is a better option than the deterministic method when partial native recall is acceptable, and when computational resources are limited. The deterministic method is a better option when 100% native recall is preferred, and a large cluster is available. Our goal was to enable scalable clone detection with classical tools using a limited number of standard workstation. Our shuffling framework satisfies this use case, which was not satisfied by a deterministic method.

11.9 IJaDataset Revisited

In Section 11.5 we used the original "core" Shuffling Framework (blind partitioning shuffling algorithm) to evaluate the ultra large IJaDataset. In Section 11.6 we discussed the deficiencies of the original technique, and in Section 11.7 we incrementally designed a technique which addresses these problems. We demonstrated the new technique's superiority by evaluating it using samples of IJaDataset where it was possible to create gold standards for all of the tools.

In this section we continue our primary experiment by using our new inverted index shuffling algorithm to evaluate IJaDataset. We compare the two algorithms in their intended use case: for big data clone detection. We limit our tool selection to Simian and NiCad. We include NiCad as the new algorithm worked best with it in the simulated experiments (Section 11.7), and because it is fast for function clone detection. We include Simian because the new algorithm showed the most conservative improvements with it, and because it is the only tool we have a gold standard for. We omit Deckard because it requires long computation times, even for smaller datasets. We do not have the computational resources to dedicate to it. We omit iClones because it did not participate in the previous IJaDataset experiment, so we can not compare the two versions of the

framework with it.

Unfortunately, prohibitive memory and execution time requirements prevented us from building gold standards for IJaDataset for any of the tools except Simian. Simian required a rented Amazon EC2 instance with 64GB of RAM and days of execution time to evaluate IJaDataset. Simian is quite fast because it only considers type 1 and type 2 clones. Renting this server for tools that have similar memory requirements, but much longer execution times, was financially prohibitive.

For these experiments we executed the inverted index shuffling algorithm for a 3-gram representation of the dataset. The similarity heuristic was parameterized to require selected file pairs to share three 3-grams. The inverted index was trimmed of any 3-grams appearing in more than 1000 files. These are the same settings used in the evaluation of the algorithm for the small test datasets. Since these settings produced good results in the test case (e.g., the framework achieved 98% total recall with NiCad), we are optimistic they are good parameters for IJaDataset. It took 12 hours to hash IJaDataset, and 40 minutes to build the 3-grams. It took 15 minutes to build the index, and 2.3 minutes to trim it. The hashed dataset only needs to be produced once and can be used with multiple executions of the shuffling framework with multiple subject tools. Changes to the dataset only require re-hashing of new or changed files.

As per the previous IJaDataset experiment, we used a maximum subset size of 50,000 files for Simian, and a maximum subset size of 10,000 files for NiCad. While NiCad could handle the 50,000 file dataset used in the evaluation of the shuffling algorithm improvements, it does not reliably in the general case, which is why a smaller subset size is used.

Recall that the inverted index algorithm executes two rounds of detection subsets. In the first round, the dataset is fully partitioned into subsets using blind shuffling. The first round parallels the original "blind" shuffling algorithm, and ensures that the clone detector is exposed to all of the intra-file clones in the dataset. The number of subsets in the first round is equal to the size of the dataset divided by the subset size for the tool (rounded up). The second round of subsets are constructed using the algorithm's new selection criteria. Pairs of files in the index which share an n-gram are selected at random and added to the subset if they satisfy the similarity heuristic and have not been previously seen together. The second round can have any number of subsets.

11.9.1 Simian

Simian detects a very large number of clone pairs in IJaDataset, more than we can process on our hardware. As with our previous IJaDataset, we instead measured our clone fragment recall heuristic. The framework's clone fragment recall for Simian is shown in Figure 11.19. For comparison, we also plot the fragment recall when the blind shuffling algorithm was used. For the first round (58 subsets), both algorithms use blind shuffling and obtain essentially identical recalls. Once the inverted index switches to its inter-file detection strategy, there is a huge difference in algorithm performance. The inverted index algorithm obtains nearly the same fragment recall within 200 subsets as the blind algorithm does in 900 subsets. In this case, the

inverted index reduces the amount of required work by nearly 80%. Interesting to note is that the inverted index algorithm provides a very quick burst of recall growth across the initial subsets, but the rate of growth quickly diminishes. It may be possible that it is beginning to reach an asymptote. This is not very desirable, and suggests that the framework may achieve a higher end recall with Simian with more relaxed file selection parameters (n-gram length, minimum similar n-grams, and inverted index trim threshold). However, we had seen some strange behavior in previous experiments with Simian. For example, the sliding effect we had previously mentioned, where Simian was reporting the same clones repeatedly with small differences in start/end lines. If the sliding effect was more pronounced when the input size was larger (e.g., the creation of the gold standard versus the detection of the subsets), it would cause a low total recall to be measured. For this reason we also decided to simulate Simian's detection of the subsets.

Also plotted in Figure 11.19 is the fragment recall results of our simulation of Simian's execution for the subsets. The simulation assumed a clone pair in Simian's gold standard was detected if the file(s) containing the clone were seen within the same subset. For measuring clone fragment recall, the cloned fragments of the detected clone pairs are also detected. In the simulated case we see much higher cloned fragment recall. This tells us that Simian fails to report some of the clone pairs in its gold standard even when the files containing these clones are shuffled together. Simian may be reporting clones inconsistently for a particular pair of files based on what other files they are input with. Or Simian's sliding effect defect may be more pronounced for a larger input size (e.g., large gold standard vs. small subset). The discrepancy between the non-simulated and simulated Simian subset detection occurs even within the first 58 subsets (round 1). During these subsets, both algorithms partition the entire dataset using blind shuffling. This suggests that Simian is not reporting many of the intra-file clones in its gold standard when it is analyzing the files in small subsets rather than all in one input. Possibly the sliding effect is more pronounced for larger input, specifically reporting many slight variations on the same fragment as a large clone class. This is consistent with casual observations we have made of Simian's gold standard.

From our analysis of Simian, we wished to estimate the general performance of the shuffling framework with the inverted index algorithm with any clone detection tool. We imagine that the performance lies somewhere between these two evaluations with Simian. The evaluation with real detection data underestimates our framework's performance as Simian is failing to detect, or is reporting differently, clones in Simian's gold standard that the shuffling framework has exposed Simian to. However, the simulation may be overestimating the recall. If Simian's sliding effect produced a large number of intra-file clones, it may be overwhelming the number of inter-file clones and boosting the recall within the 58 subsets higher than if Simian did not have this defect.

The framework was executed on a solid state drive which provided greatly improved execution time of both the shuffling framework and Simian. Building the detection subsets from their specification (i.e., assembling the files for detection) took less than 5 minutes per subset. Choosing the files for the subsets of round 1 (blind shuffling) took less than 1 second per subset. For round 2, subset generation time started at 30 seconds

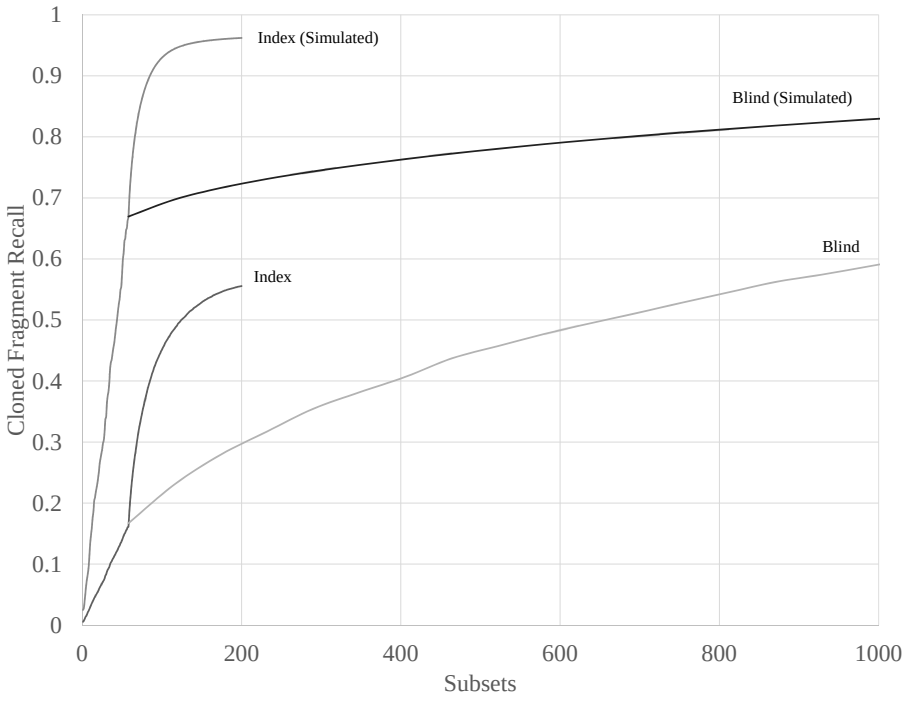


Figure 11.19: Index vs. Blind Shuffling Algorithm For IJaDataset Using Simian

per subset and increased as more subsets were generated up to 44 minutes per subset by the 142th subset in round 2 (200th subset in total), with a total generation time of 24 hours. From Section 11.7's study of the inverted index algorithm, we saw that when subset generation time had increased by 2 orders of magnitude, that the framework had mostly exhausted its search space, and reached its maximum total recall. Subset assembly took on average 1.33 minutes per subset. Simian's execution time per subset was 1.28 minutes on average, with a range of 7 seconds to 12 minutes, and a total execution time of 5 hours. For Simian, subset generation time exceeds execution time. This is expected as Simian's scalability limit for big data is not execution time, but its memory requirements.

11.9.2 NiCad

Like the previous IJaDataset experiment, we evaluated the shuffling framework's performance with NiCad by measuring the number of unique detected clone pairs and cloned fragments across the subsets. We could not measure total recall as NiCad cannot be scaled to IJaDataset even with extraordinary hardware. It has internal limitations that restrict the size of the input in terms of the number of source lines and the amount of cloned code. Even if these limitations are removed, and given sufficient RAM, it could take months of execution time to produce the gold standard.

In Figure 11.20 we show the shuffling framework's cumulative detection of unique clone pairs across the subsets using NiCad. We also show the detection performance using the blind shuffling algorithm for

comparison. The blind algorithm data is taken from the previous IJaDataset experiment (Section 11.5). The detection rounds for both algorithms are indicated by the circle markers.

For the first round, both algorithms use blind shuffling and have nearly identical clone pair detection performance. Once the index algorithm begins its second round, a considerably better clone pair detection performance is observed. The index algorithm is able to detect approximately the same number of clone pairs in 438 subsets as the blind algorithm does in 5780 subsets (20 "blind shuffling" rounds), a 92% reduction in subsets. The number of subsets needed to achieve the same result is reduced by a whole order of magnitude. This is a considerable decrease in the number of required clone detection tool executions.

In Figure 11.21 we show the shuffling framework's cumulative detection of unique clone fragments across the subsets using NiCad. Again, for the first round where both algorithms use blind shuffling, the detection performance is essentially identical. Like with the clone pairs, we see a large improvement in cloned fragment detection using the index algorithm over the blind algorithm. The index algorithm detected approximately the same number of cloned fragments within 299 subsets as the blind algorithm did in 5780 subsets, a 95% reduction in subsets. The index algorithm finds nearly double the cloned fragments within 488 subsets as the blind algorithm does in 5780 subsets. Considering only the clone pairs detected after the first round (where both use blind shuffling), the index algorithm detects 3.5x the clone pairs in 200 subsets as the blind algorithm does in 5493 subsets.

The improvement in cloned fragment detection with the index algorithm is larger than that of the improvement in clone pair detection. Unlike with the clone pairs, we see a noticeable decay in the growth of detected cloned fragments. The average number of new cloned fragments detected per subset is decreasing, suggesting that they are becoming rarer. This suggests that the cloned fragments are being found faster than the clone relationships between them. A transitive clone recovery technique could be used to recover these missing relationships without additional subsets. Since the index algorithm is detecting the cloned fragments much faster than the blind algorithm, the transitive recovery technique would be even more valuable when used with the index approach. Before this is possible, an efficient and precise way to apply transitivity to type 3 clones, for which the validity of transitivity would need to be checked in each instance, needs to be devised.

The framework was executed on a solid state drive which greatly improved the execution time of the shuffling framework and NiCad. Building the subsets after their file contents had been chosen took 0.25 minutes on average. Choosing the files for the subsets of the first round (blind partitioning, subset 1-258) took 60ms on average. For round 2 (inverted index, subsets >248), and consistent with our framework improvement study, subset generation time started short (a few seconds) and grew as more subsets were generated, to a couple minutes by subset 1100. NiCad's execution time per subset was 3.5 minutes on average, with a range of 1.3-15.6 minutes.

Subset generation time is plotted in Figure 11.22. The gray line shows the subset we stopped executing NiCad at for this experiment. Between subset 259 (the start of the inverted index algorithm) and subset

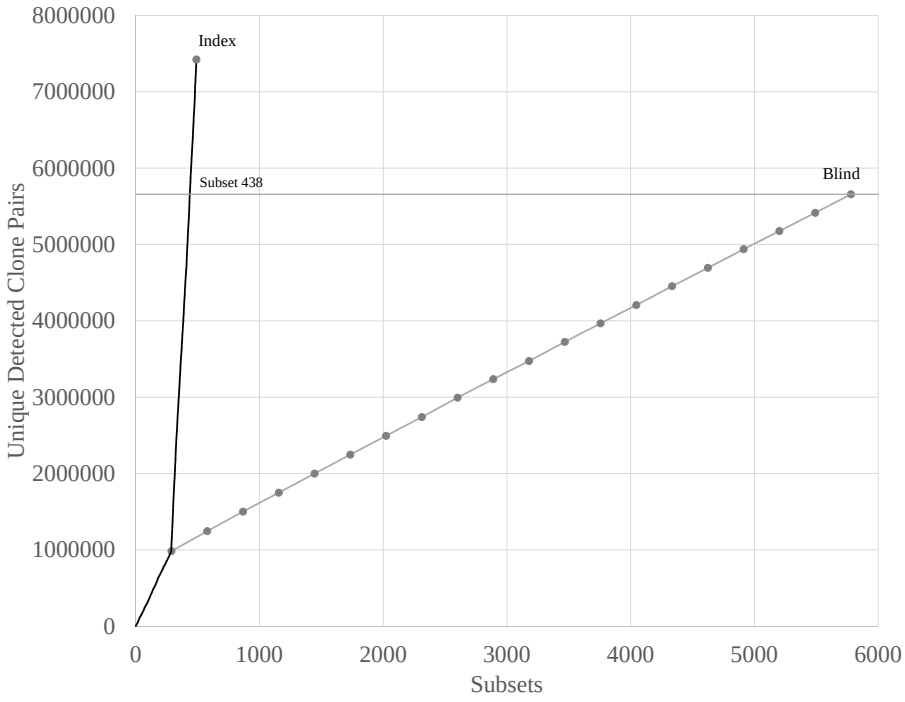


Figure 11.20: Index vs. Blind Shuffling Algorithm Clone Pair Detection For IJaDataset With NiCad

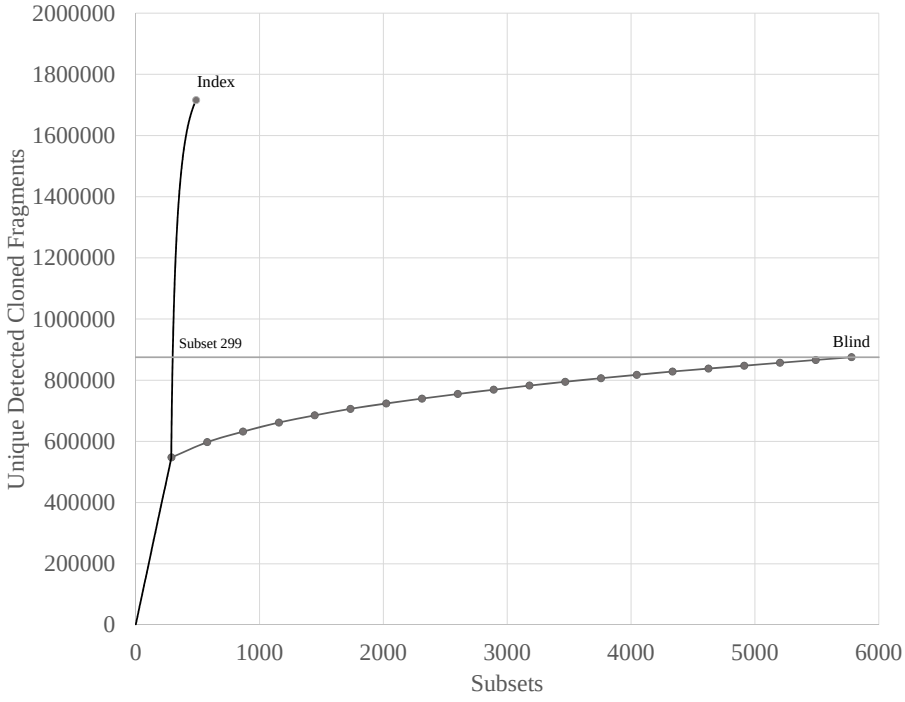


Figure 11.21: Index vs. Blind Shuffling Algorithm Clone Fragment Detection For IJaDataset With NiCad

1200, the subset generation time increases by an order of magnitude. When we tested the inverted index algorithm with NiCad and a dataset of 50,000 files, we found that total recall had reached 90% by the time that the subset generation time had increased by an order of magnitude. Since we used a similar ratio between subset size and dataset size in the test experiment as we have in this IJaDataset experiment, perhaps the shuffling framework would achieve 90% total recall of NiCad's gold standard for IJaDataset by round 1200. Unfortunately, we can not verify this as it is not practical to compute NiCad's gold standard for IJaDataset.

With the inverted index algorithm, the subsets must be generated serially, since the contents of a subset depend on the contents of previous subsets. This is a potential bottleneck if the execution of the tool for these subsets is distributed over a number of computers. In Figure 11.23 we plot the time at which each subset is ready for evaluation. This is the cumulative subset generation time versus subset. Alongside this we plot the time at which NiCad's evaluation of a subset is complete when 1, 2, 4, 8, 16 or 32 computers are utilized. Time is counted in minutes from when the generation of the first subset of round 1 began. For this calculation we considered a computer occupied for 3.75 minutes to evaluate a subset, which includes the average NiCad execution time for a 10,000 file subset, and the average time required to assemble a subset from its specification (a file list). We assume that IJaDataset is on each computer, and that it takes negligible time for a subset specification to be sent to a computer. Since we are assuming a uniform NiCad execution time, we consider every nth subset to be sent to a specific computer, where n is the number of computers. For example, with 4 computers in a cluster, subsets 1, 5, 9, ... goes to computer one, subsets 2, 6, 10 go to computer two, etc. Therefore, NiCad's analysis of a subset is complete exactly 3.75 minutes after the later of: (1) the time the subset's generation was complete, or (2) the time at which the computer finished analyzing its previous subset. Case (1) will only happen when a computer is waiting for its first subset, and once generation time becomes a bottleneck and the computer is idle waiting for its next subset to be generated.

From this distributed execution estimate, we do not see 1, 2 or 4 computers becoming bottle-necked by subset generation within the 1200 subsets we generated. 8 computers become bottle-necked after 1080 subsets, after which at least one computer is idle. This occurs at subset 854 with 16 computers, and subset 664 with 32 computers. The intention of this framework was to enable the scaling of classical tools on standard hardware. This plot shows us that the shuffling framework's subset generation time will not bottleneck a budget compute cluster of 2-4 computers. The bottleneck may occur at a later subset when the framework is used with other tools, or even different configurations of NiCad, that require longer execution times. For this experiment we executed NiCad in its most basic configuration. Since NiCad was only looking for function granularity clones, and did not perform any normalization beyond pretty printing, its execution time for the 10,000 file subsets was quite fast for type 3 detection. Tools which look for clones at lower granularities will likely have longer execution time. NiCad's execution time also grows considerably when its advanced features are enabled.

The bottleneck could be overcome by generating multiple subsets simultaneously when a computer in the cluster is idle. When determining if a randomly selected pair of files (that satisfy the similarity heuristics) is

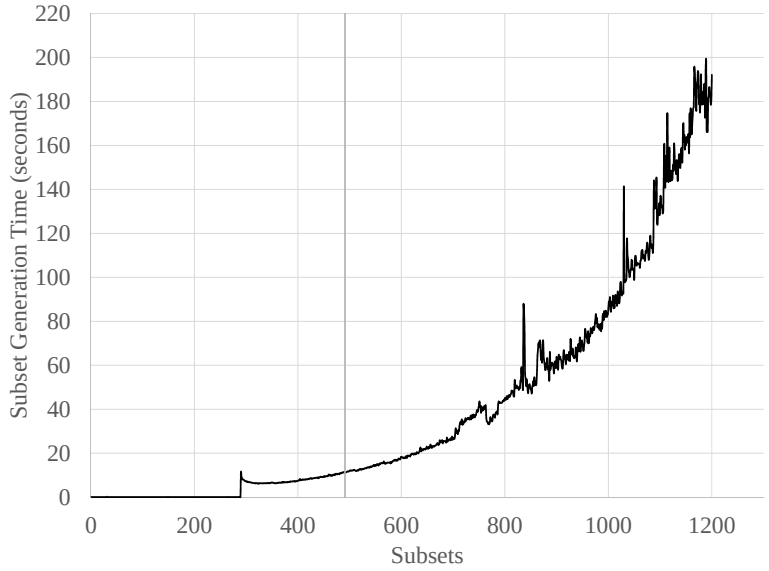


Figure 11.22: Subset Generation Time - 10,000 file subsets of IJaDataset - NiCad

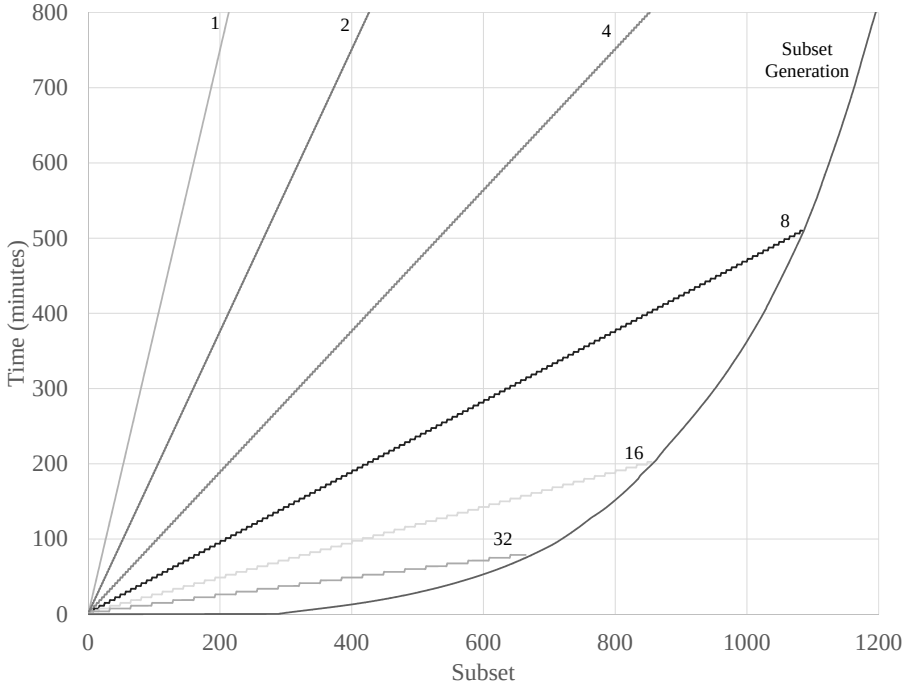


Figure 11.23: Subset Generation Time - 10,000 file subsets of IJaDataset - NiCad

unseen, the algorithm would consult the contents of the previously generated subsets, but not the contents of other subsets currently being generated. As such, some of the same file pairs may be selected for subsets generated at the same time. However, the probability of this would be low unless total recall was very close to 100%. This technique would ensure that all the computers in the cluster are continuously utilized. We will explore such a scheme as part of our future work towards a publicly released tool version of this framework.

11.9.3 Summary

In summary, we experienced considerable gains in detection performance with the inverted index shuffling algorithm over the blind shuffling algorithm when evaluating IJaDataset using Simian and NiCad. With Simian we found that the index algorithm allows a higher cloned fragment recall to be obtained with fewer subsets. With NiCad we found that the index algorithm was able to match the blind algorithm using an order of magnitude less subsets. From these results we conclude that the inverted index algorithm greatly exceeds the performance of the blind shuffling algorithm, **RQ#5**.

11.10 Conclusion

In this research we presented and demonstrated the shuffling framework for scaling classical clone detection tools to big data on standard consumer-level (i.e., affordable) workstation-class hardware. The shuffling framework scales classical tools by executing them for non-deterministically chosen subsets of a big data source dataset. We began with the version of the shuffling framework we proposed in previous work [64], which we termed the "core shuffling framework" which used the "blind partitioning shuffling algorithm". We evaluated this version of the framework using ordinary sized systems (for comparison against gold standards) and for its application to real big data (IJaDatase 2.0). While this version of the framework successfully scaled the classical tools to big data, the execution time required to obtain a satisfactory ratio of a tool's native recall was still high. We used these experiments to identify the deficiencies in the approach. Specifically, the blind partitioning algorithm did not prevent the same files from being randomly shuffled together repeatedly, and it did not consider the similarity of the files it was shuffling together.

Considering these deficiencies, we iteratively improved the shuffling framework by modifying the original shuffling algorithm. We explored methods of tracking files that have been seen by the tool previously (to prevent shuffling them together repeatedly), as well as n-gram and inverted index based file similarity heuristics (to prevent shuffling together of dissimilar files). We evaluated the improvements using a sample of IJaDataset small enough to evaluate the subject clone detectors' gold standards. We evaluated the improvements using the same subset size to dataset size as used in the big data case with IJaDataset. We termed our final algorithm the "inverted index shuffling algorithm". We found from our evaluations that this algorithm was able to scale clone detectors to big data while capturing up to 90-95% of the a clone detector's native recall without sacrificing its precision. We then applied our new algorithm to the big data IJaDataset,

and found that it was able to capture the detection performance of our original "blind partitioning shuffling algorithm" using 90% fewer subsets of IJaDataset, thereby improving our frameworks scalability by an order of magnitude.

Using our approach, classical clone detectors can be used to detect clones in big data on commodity hardware. Researchers and developers can use their familiar, available, proven and well-understood classical tools to build clone corpora for ultra-large inter-project software datasets. These corpora may be used to study developer behavior within a corporation or globally (open-source). Duplicated engineering efforts in open-source or within a corporation can be reduced by extracting the duplication found into new software libraries. Large corpora can be used within Internet-scale clone search to provide API recommendation and usage support. Our approach comes at the cost of a fraction of a tool's native recall. However, a good clone corpora is built using multiple scalable and classical tools. In this intended use case, lower native recall is made up for by the consultation of multiple and varied clone detectors.

Chapter 12

CloneWorks: Fast, Scalable and User-Guided Clone Detection for Large-Scale

One of the most active topics in clone research is the detection of clones within big inter-project source code datasets containing on the order of thousands of software projects or more. This has many potential applications, including: studying global open-source practices [96], mining the seeds for new APIs [48], license violation detection [73], similar mobile application detection [22], large-scale clone and code search [61, 81], code completion [49], API recommendation and usage support [66], and so on. These applications require clone detectors that scale to hundreds of millions of lines of code or larger. Large-scale clone detection is also needed for software product line migration [45], and for clone detection in growing industrial software portfolios which are reaching millions [3] or even billions [91] of lines of code.

In order to achieve these emerging applications, fast, scalable and user-guided clone detection tools are needed. While a number of scalable tools and techniques have been published [22, 39, 44, 47, 48, 61, 62, 73, 84, 116, 126] they have a number of limitations. Many are domain-specific and designed for particular use-cases [22,39,62,73,81], and are not suitable for other purposes or for general detection. Many of the techniques only support Type-1 and Type-2 clone detection [47, 48, 73, 84], where only minor editing changes occur in the copy and pasted code fragments. Type-3 clones, where further editing such as the addition/removal and modification of statements are made between the copied code fragments, are the most challenging to detect in terms of technique and especially scalability. Some require extraordinary hardware, in particular large amounts of memory [44], or distribution across a compute cluster [39, 47, 84], which can be costly and difficult to setup. We want large-scale clone detection to execute and scale on even an average workstation (e.g., i7 CPU, 12GB RAM, SSD) that is accessible and affordable to researchers with minimal setup and administrative overhead. The execution time of even the fastest near-miss tool in the literature to scale on an average workstation [116] requires days of execution time for a big inter-project dataset. None of the existing tools can guarantee scalability to any input size within typical memory constraints.

While the existing scalable tools use various source normalizations to improve clone detection, they are often not user-guided in their configuration [106]. Researchers would benefit from a user-guided clone detection tool that lets them customize the source normalization and source representation for clone detection, including the insertion of custom source transformation code, without needing to re-implement a whole new

parser or a completely new clone detection tool. The emerging applications and research of clones in large inter-project repositories require this domain-specific pre-processing and representation of the source code before clone detection. This could be to find particular types of clones, to target clones with particular kinds of differences or patterns, or to explore novel kinds of clones such as API usage clones. A tool that enables this with a minimum of effort is needed by the community [106, 126].

In this chapter, we present CloneWorks, a fast, scalable and user-guided clone detector. As a user-guided detector, it gives the user fine-grained control over the normalization, transformation, processing and representation of source-code for clone detection. This way the user can decide on the types of clones they would like to detect, as per their scenario or use-case (e.g., API clones). Clone detection is achieved using a fast and efficient modified Jaccard similarity coefficient [67, 116], which represents each code fragment as a set of terms, for some user-controlled term definition. Scalability in execution time is achieved using the sub-block filtering optimization [116] with clone indexing. We achieve fast execution time by keeping the index and the indexed code fragments in-memory in low-complexity data structures. Scalability within memory constraints, regardless of the input size, is achieved using a targeted input partitioning scheme. CloneWorks supports the detection of block, function and file clones in Java, C, and C# source code. CloneWorks is designed to scale on an average workstation, but can also scale up to any number of cores or amount of memory on extraordinary hardware.

We have extensively evaluated the performance of CloneWorks, and compared it against eight competing tools. We measure recall using a large collection of real clones within 25K software projects (BigCloneBench [122, 125, 129]). We also measure recall in a controlled experiment using synthetic clones produced by the Mutation and Injection Framework [111, 131]. We measure precision by manually validating 400 random clones reported by each tool during the BigCloneBench experiment. We evaluate the scalability and execution time of CloneWorks and the competing tools for inputs up to 250MLOC. From our evaluations, CloneWorks emerged as the state of the art in general clone detection, with the best recall and precision for Type-1, Type-2 and Type-3 clones. CloneWorks has the best scalability and execution time for large inputs, and completes Type-3 clone detection on a 250MLOC input in just 2-10 hours, depending on the configuration, on an average workstation.

We evaluated the user-guided aspect of CloneWorks by exploiting the different normalization and customization options of CloneWorks on IJaDataset [4], an inter-project dataset with 25K systems, to see how CloneWorks could benefit the users in guided detection. We found that the user-guided approach allows targeted detection of clones with particular features that may otherwise be missed by traditional detection procedures. As part of this evaluation, we manually validated 16K clone pairs detected by CloneWorks under different configurations, and found good precision with the user-guided approach. To our knowledge, this is the most extensive precision evaluation for any tool to date.

In summary, this chapter makes the following contributions:

- A fast and scalable clone detector. Scales to 250MLOC in just 2-10 hours, depending on the configu-

ration, which is an order of magnitude faster than the state of the art.

- A user-guided input converter for clone detection, with various transformations and including a plug-in architecture.
- Extensive evaluation and comparison with BigCloneBench and Mutation Framework clone benchmarks. CloneWorks has the top recall and precision of the state of the are tools.
- A demonstration of the user-guided aspect with scenarios and case studies performed on a large inter-project dataset containing 25K projects, including the validation of 16K detected clone pairs.

CloneWorks has been previously published as a tool [130] and poster paper [123]. This chapter is based on a manuscript that significantly extends beyond the published versions and is currently under blind review at a top tier conference. The manuscript has been reformatted to fit this thesis.

This chapter is organized as follows. The CloneWorks approach is shown in Section 12.1, we explain how it achieves fast clone detection in Section 12.2, its approach to scalability in Section 12.3, and its user-guided aspects in Section 12.4. Section 12.5 contains our evaluation of CloneWorks, including its comparison against the competing tools, and an extensive exploration of its user-guided features. We discuss the limitations of our work in Section 12.6, compare it to the related work in Section 12.7, and summarize the contributions of CloneWorks in Section 12.8.

12.1 The CloneWorks Approach

CloneWorks has two major components: the user-guided *input converter* and fast and scalable *clone detector*, which are shown in Figure 12.1.

The user-guided *input converter* is used to prepare the source code for clone detection. This includes parsing the input source code files, extracting the code fragments of a given granularity, and converting their source text into a set of terms representation for clone detection. Users customize this set of terms representation by specifying the pretty-printing, normalizations, transformations, abstractions and filtering applied to the code fragments, and how the code fragments should be split into terms. This can be as simple as splitting the code fragments into their language tokens or source lines, or could be as complex as converting the code fragments into the set of API call patterns they contain. Users can select from a number of included source processors that can be applied at the source or term levels, or provide their own by a plug-in architecture. In this way, users can target novel experiments with a minimum of implementation effort, while taking advantage of our efficient and parallel parsing and transformation architecture. We discuss this in detail in Section 12.4 and Section 12.5.6.

The *clone detector* receives the code fragments as term sets from the input builder, and evaluates every pair of code fragments with the modified Jaccard similarity metric (Eq. 12.1), reporting those satisfying a minimum similarity threshold as clones. The sub-block filtering optimization [116] is used to avoid comparing code fragments that cannot satisfy the given minimum similarity threshold. Clone indexing [47,116] is used to

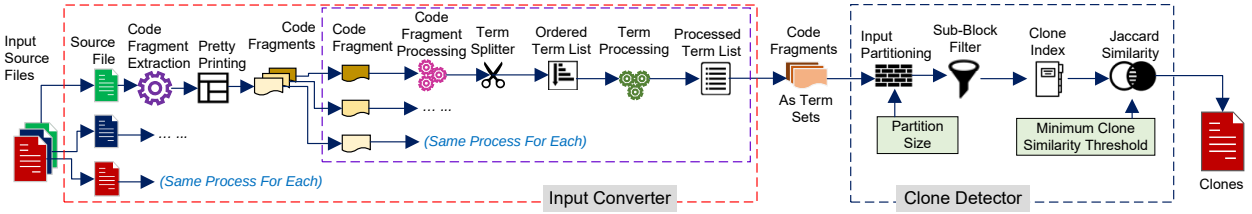


Figure 12.1: The CloneWorks Approach

quickly identify the potential clones for comparison. We achieve very-fast execution time by storing the code fragments and index in low-complexity in-memory data structures, and then scale within memory constraints using a novel input partitioning scheme. We discuss the clone detector further in Sections 12.2 and 12.3.

12.2 Fast Clone Detection

CloneWorks detects clones using a modified similarity coefficient, which is shown in Eq. 12.1. It takes a pair of code fragments, f_1 and f_2, as the set of terms (e.g., tokens, statements, API calls, etc.) they contain, including duplicates, and measures their similarity as the minimum ratio of the intersection of their terms. A pair of code fragments is reported as a clone if their similarity exceeds a given threshold (e.g., 70%). The code fragments are stored as hash sets over the terms (as strings) they contain, allowing the modified Jaccard coefficient to be measured in $O(m)$ time, where m is the size of the larger code fragment. An upper and lower bound on the similarity is tracked during the computation for early rejection or acceptance of a potential clone pair. Despite the simplicity of this metric, and the fact that it ignores the original ordering of the code terms, we find it can achieve both high recall and precision (Section 12.5).

$$sim(f_1, f_2) = \frac{|f_1 \cap f_2|}{max(|f_1|, |f_2|)} = min(\frac{|f_1 \cap f_2|}{|f_1|}, \frac{|f_1 \cap f_2|}{|f_2|}) \tag{12.1}$$

Clone detection is then the measurement of similarity for every distinct pair of code fragments in the input, $I = \{f_1, f_2, ..., f_n\}$, and collecting those that satisfy a minimum clone similarity threshold, as shown in Eq. 12.2. The code fragments are prepared by the input converter (Figure 12.1 and Section 12.4), and the clone detector receives them pre-formatted as term sets. The clone detection process is then independent of the language and granularity of the code-fragments, the transformations applied, and the term definition.

$$D(I) = \{(f_i, f_j) \; \epsilon \; I \times I \mid i < j \wedge sim(f_i, f_j) \geq t\} \tag{12.2}$$

We use the *sub-block filtering* optimization [116] to skip the comparison of many code fragment pairs that cannot possibly satisfy a minimum similarity threshold. Consider potential clone pair (f_1,f_2), and let l_1 and l_2 be their lists of terms, including duplicates, sorted in a definite term order. In order to satisfy a minimum similarity threshold t, f_1 must share at least $\lceil t\,|f_1| \rceil$ terms with f_2, and vice-versa. Then, if

$sim(f_1, f_2) \geq t$, it must also be true that the prefix of l_1 of length $|f_1| - \lceil t\,|f_1| \rceil + 1$ and the prefix of l_2 of length $|f_2| - \lceil t\,|f_2| \rceil + 1$ must share at least one term. Code fragments whose prefixes do not share at least one term do not need to be compared as they cannot satisfy the similarity threshold. This optimization significantly reduces the number of code fragments needing to be fully compared when terms are ordered by increasing global-term-frequency [116]. Clone detection can then be performed as shown in Eq. 12.3, where F is the sub-block filter and $F(I \times I)$ is the set of code fragment pairs that share at least one term in their prefixes.

$$D(I) = \{(f_i, f_j)\ \epsilon\ F(I \times I) \mid i < j \wedge sim(f_i, f_j) \geq t\} \tag{12.3}$$

We efficiently identify the code fragment pairs that could be clones, as determined by the sub-block filter, using an inverted clone index [47, 116, 126]. Each code fragment is indexed for its prefix terms. A given code fragment then only needs to be compared against the set of code fragments returned when the index is queried for each of the query code fragment's prefix terms. Clone detection is complete once the index is queried for each of the code fragments, and the resulting potential clone pairs evaluated with the similarity coefficient. $F(I \times I)$ in Eq. 12.3 is then efficiently computed by querying the index with each code fragments in I. We use an in-memory index (simple hash map) for fast O(1) access.

Clone detection proceeds in three phases. (1) The code fragments, as transformed into terms sets by the input converter, are read into memory, and their global term frequencies are computed. (2) The code fragments' term sets are sorted by increasing term frequency, their prefixes are computed, and they are indexed for only their prefix terms. (3) The index is queried by each code fragment's prefix terms, and the resulting potential clone pairs are evaluated by the similarity coefficient, with those exceeding the given similarity threshold output to a detection report file. Each of the phases are parallelized, and can scale to any number of available threads. To make this very fast, the index is stored in-memory as a hash map for O(1) access and the code fragments are kept in-memory for immediate access when returned from the index.

12.3 Scalable Clone Detection

We achieve scalability in execution time using the modified Jaccard similarity coefficient, which is a linear computation, and by reducing the number of code fragment comparisons using the sub-block filtering optimization. We achieve very fast clone detection using a purely in-memory approach. Specifically, we store our clone index and code fragments in-memory in data structures prioritizing execution performance at the cost of high memory requirements. We then scale in available memory by partitioning the input code fragments and creating a separate index for each partition. Clone detection is then executed for each partitioned index separately, such that only the current partition's index and its indexed code fragments need to be held in memory at a given time. Each partitioned index is queried for all of the code fragments, which returns all potential clone pairs that include at least one code fragment from that partition, subject to sub-block

filtering. Clone detection with a particular partitioned index detects all the clones involving at least one code fragment from that partition. Across all of the executions, all of the potential clones have been investigated, and those satisfying the similarity threshold reported as clone pairs. Clone symmetry is exploited: given k partitions, index p only needs to be queried for the code fragments in partitions p through k.

For each execution, only the code fragments within the current partition need to be held in memory to ensure $O(1)$ access when queried from the index. The code fragments from the other partitions, which are used to query the index, can be streamed in, but do not need to be retained in memory. This incurs some overhead, as the code fragments not in the current partition must be streamed from the disk for querying the index. This is minimal, with each code fragment having to be loaded into memory on average $\frac{k+1}{2}$ times. Streaming the code blocks is predictable and sequential IO, so the upcoming code fragments can be preloaded as to not stall the clone detection threads.

Our partitioning approach maintains the speed of our computational efficient but memory intensive implementation, while scaling within even conservative memory constraints. This approach is formally specified in Eq. 12.4. The code fragments are split into k non-overlapping partitions: $I = \{I_1 \cup I_2 \cup ... \cup I_k\}$, and an index is created for each partition: $H = \{H_1, H_2, ..., H_k\}$. The clone detection results are then the union of our clone detection approach applied for each partitioned index. $F_p(I_{p:k} \times I_p)$ is sub-block filtering applied on all potential clone pairs between the code fragments in partitions p through k and the code fragments in partition p using index H_p. This is $I_{p:k} \times I_p$ instead of $I \times I_p$ as symmetry can be exploited.

$$D(I) = \bigcup_{p=1}^{k} \{(f_i, f_j) \; \epsilon \; F_p(I_{p:k} \times I_p) | i \leq j \wedge sim(f_i, f_j) \geq t\} \tag{12.4}$$

12.4 User-Guided Clone Detection

The user-guided input converter is responsible for extracting code fragments from the input source files and converting them into sets of terms representations for the clone detector. We call this user-guided because the user has full control over this process, including the plug-in of custom source-code and term processing logic. Users can configure the input builder to produce a code fragment representation for targeting specific clone types, or for a novel kind of clone as needed for their experiment. We discuss here the user-guided procedure, and then discuss how it can be exploited, including empirical case studies, in our evaluation of the user-guided approach in Section 12.5.6.

As shown in Figure 12.1, each source file is parsed and the code fragments of a specified granularity are extracted and pretty-printed. Then, for each code fragment in that file, the following processing occurs. First, a number of user-specified code-fragment processors are applied to the code fragment, which can apply normalizations and transformations to the source syntax. Then, the terms are extracted by term splitting, which outputs the code fragment as a list of the terms it contains, including duplicates, in the order of their occurrence. The term list is then processed by a user-specified sequence of term processors, which

take a term list as input and output the same list with some specified modifications, such as term filtering, splitting, combining, transformation, and so on. The term list is then reformatted to a set of terms, including duplicates, as expected by the clone detector. The prepared code fragments from all of the input source files are collected.

Extraction and Pretty Printing: The source file is parsed into a language-specific abstract syntax tree (AST), and code fragment sub-ASTs of the specified granularity are located and extracted. De-parsing the code fragment ASTs also applies a strict pretty-printing and removes any comments. The input builder supports the extraction of block, function and file granularity code fragments in Java, C, and C# source files, and could be extended to additional languages and granularities. The pretty-printing can be customized to change how the code fragments are formatted for splitting.

Code Fragment Processors: The user specifies a number of code-fragment processors to be applied, in their specified order, to the code fragments. These can be used to layout, format, normalize or transform the code fragments. They can also be used to remove code fragments from consideration, or generate multiple code fragments (e.g., alternate normalizations) from a single code fragment. The processors are registered with the input builder as executables along with their configuration parameters. The processors receive the language and granularity of the code fragments as parameters, and are expected to take the code fragments as input, and output the transformed code fragments in a simple format. We include a number of processors for identifier normalization, syntax element abstraction and filtering, which we describe in more detail in Section 12.5.6. Users can provide their own processors, implemented in any language or technology, by providing a compliant executable.

Term Splitting: The input builder supports splitting by source line or by language token. By default, splitting by source line results in code-statements as terms, since the strict pretty-printing results in one code statement per line. The pretty-printing can be customized, or source transformations can be used, to layout the code differently in order to customize the term definition when splitting by line. Splitting by language-token is done using a language-specific token grammar, which splits the source-code into its keywords, literals, identifiers, separators, operators, and so on.

Term Processors: Term processors allow the user to transform the code fragments at the term level. Term processors take the list of terms in a code fragment as input and output the term list after some modification. The term processor may be used to add, split, combine, filter, transform, and so on, the terms based on some conditions. Term processors can also be used to filter a code fragment from consideration based on the analysis of its terms. The user can specify any number of term processors, which are applied in their specified order. We describe some of the included term processors in Section 12.5.6. Users can add new term processors, by implementing the term processor interface, which are discovered and configured at runtime.

Implementation: The input builder creates n threads with the parsing, code-fragment, and term processor pipeline indicated by the user. Each thread processes a different source file from the input in parallel.

Intermediate results are passed along the pipe-line in-memory, instead of written to disk, to avoid unnecessary and slow IO operations. A separate thread pre-loads upcoming source files into memory, while another thread collects the final results and writes them to disk, as to not block the parsing threads.

12.5 Evaluation

In this section, we empirically evaluate the performance of CloneWorks, and compare it against eight well established clone detection tools, including: CCFinderX [58], CtCompare [133], Deckard [53], iClones [41], NiCad [110], SimCad [136], Simian [44] and SourcererCC [116]. We compare the tools for their recall, precision, execution time and scalability. We conduct our evaluation using the recent clone benchmarks: BigCloneBench [122, 125, 129] and the Mutation and Injection Framework [111, 131]. We evaluate CloneWorks for the three configurations shown in Table 12.1. We try a simple line-based approach, a token-based approach, and a pattern-based approach where we add identifier and literal normalizations to the line-based configuration. We evaluate execution time and scalability using IJaDataset [4], a large inter-project Java dataset containing 250MLOC from 25K projects [122, 125, 129]. We only use Deckard in the C experiment, as its stable version does not support C# and modern Java syntax for parsing. We empirically demonstrate CloneWorks's user-guided approach by example and through case studies exploiting custom transformations using the input converter.

12.5.1 Mutation Framework

We measured the recall of CloneWorks and the competing tools using our Mutation Framework (Chapter 3). We used the framework to produce clone benchmarking corpora for Java, C and C# clones. We synthesized clones using 250 randomly selected functions, the 15 mutation operators, and 10 randomly selected injection locations per clone, for a total of 37,500 unique reference clones per language (112,500 total). We used IPScanner (Java), Monit (C) and MonoOSC (C#) as the subject systems. We restricted the clones to 15-

Table 12.1: Tool Configurations for Mutation Framework and BigCloneBench Experiments

Tool	Mutation and Injection Framework	BigCloneBench
CloneWorks (T3-Line)	15+lines, 70% similarity threshold, split by line	10+ lines, 70% threshold, split by line
CloneWorks (T3-Token)	15+ lines, 70% similarity threshold, split by language tokens, filter operator & separator tokens.	10+ lines, 70% similarity threshold, split by language tokens, filter operator & separator tokens.
CloneWorks (T3-Pattern)	15+ lines, 70% similarity threshold, arbitrary identifier-renaming, literal abstraction, split by code-statements.	10+ lines, 70% similarity threshold, arbitrary-identifier-renaming, literal abstraction, split by code-statements.
CCFinderX	100+ tokens, 12+ token types, soft block shaper.	100+ tokens, 20+ token types, hard block shaper.
CtCompare	100+ tokens, max. 3 isomorphic relations.	100+ tokens, max. 6 isomorphic relations.
Deckard	Min length 100 tokens, 85% similarity, 4 token stride	-
iClones	100+ token clone size, 20+ token blocks.	90+ tokens clones, 20+ tokens.
NiCad	15+ lines, blind-renaming, abstract-literal, max. 30% dissimilarity.	10+ lines, blind-renaming, abstract-literal, max. 30% dissimilarity.
SimCad	15+ lines, unicode support, greedy transformation.	10+ lines, unicode support, greedy transformation.
Simian	15+ lines, ignore: identifiers, literals.	10+ lines, ignore: case, subtype names, modifiers.
SourcererCC	15+ lines, 70% similarity threshold.	10+ lines, 70% similarity threshold.

Table 12.2: Recall Per Clone Type and Precision Results

Tool	Mutation and Injection Framework									BigCloneBench					
	Java			C			C#			Java					
	T1	T2	T3	T1	T2	T3	T1	T2	T3	T1	T2	VST3	ST3	MT3	Precision
CloneWorks (T3-line)	100	99	100	100	99	100	100	98	100	100	97	90	40	0	100
CloneWorks (T3-token)	100	100	100	100	100	100	100	100	100	100	99	92	65	7	95
CloneWorks (T3-pattern)	100	100	100	100	100	100	100	100	100	100	99	98	92	3	86
CCFinderX	99	77	0	100	83	0	100	83	0	99	93	34	2	0	76
CtCompare	96	48	0	69	40	0	-	-	-	92	86	7	1	0	17
Deckard	-	-	-	73	72	69	-	-	-	-	-	-	-	-	-
iClones	100	93	96	100	96	99	-	-	-	98	90	30	5	0	89
NiCad	100	100	100	99	99	99	98	98	98	100	99	98	92	0	87
SimCad	100	96	89	100	97	89	100	97	88	100	99	89	47	7	16
Simian	81	90	0	85	97	0	75	43	0	65	27	0	1	0	52
SourcererCC	100	100	100	100	100	100	100	100	100	100	99	92	63	4	93

200 lines and 100-2000 tokens in length, and a mutation containment of 15%. We prefer a larger clone size here because executing some of the tools for a small clone size hundreds of thousands of times is very time intensive. Previous studies [122,128] show that this configuration gives accurate results. To successfully detect a reference clone, the tool must reported a clone that subsumes 70% of the reference clone, and handles the clone-type specific edit introduced by the operator. The configurations of the tools for this experiment are found in Table 12.1. We configured the tools to use all of their source normalization features.

Recall measured by the Mutation Framework is summarized in Table 12.2. For brevity, we summarize recall per clone type by averaging across the mutation operators that produce clones of that type. The three configurations of CloneWorks have perfect (100%) or near-perfect (98-99%) recall for all three clone types. This shows that CloneWorks can handle any of the types of edits developers make in copy and pasted code of the first three clone types. SourcererCC and NiCad have similar performance, while the other tools have weaknesses for some of the clone types.

12.5.2 Recall - BigCloneBench

In this section, we measure the recall of the tools using a real-world benchmark, BigCloneBench (Chapter 8). [122,125,129]. We observe how the capabilities of the tools as measured by the Mutation Framework translate into real-world performance for real clones produced by real developers. We use the BigCloneEval [129] release of the benchmark, which contains the latest version of BigCloneBench with 8 million clones of 43 distinct functionalities, and a tool for performing recall measurement experiments.

We configure BigCloneEval to evaluate recall for clones that are 10 lines and 50 tokens in length or longer. While 6 lines is common for benchmarking [13,122,128], this is a small clone size for detection in large inter-project source repositories [116,126]. We use 10 lines as a realistic minimum clone size in a large-scale clone detection experiment [116,126,126]. Recall was measured using a coverage-based clone matching algorithm with a 70% threshold. To successfully detect a reference clone, the clone detector must report a clone that

covers 70% of the reference clone by source line.

BigCloneEval measures recall per clone type. Since Type-3 and Type-4 clones span a significant region of syntactical similarity, it divides these into discrete similarity regions. Very-Strongly Type-3 (**VST3**) clones are those with syntactical similarity in the range 90% (inclusive) to 100% (exclusive), Strongly Type-3 (**ST3**) in 70-90%, Moderately Type-3 (**MT3**) in 50-70%, and Weakly Type-3/Type-4 (**WT3/T4**) in 0-50%. Syntactical similarity is measured by line after Type-1 and Type-2 normalizations [125]. We disregard the WT3/T4 category as the participating tools are syntax-based and do not have meaningful detection in this category.

We executed the tools for BigCloneBench using empirically chosen configurations, which we summarize in Table 12.1. We explored the configuration options of the tools, and found good configurations by evaluating the tools many times while varying their parameters. We choose as our final configurations those that maximize recall while balancing precision. We could measure recall for each configuration using BigCloneBench, but measuring precision for each configuration would require too extensive of manual validation. We estimated the effect of a configuration change on precision by considering the increase in the number of detected clones relative to the change in recall.

Recall achieved by the Type-3 configurations of CloneWorks, and the configurations of the competing tools, is summarized per clone type in Table 12.2. With its three Type-3 configurations, CloneWorks has perfect Type-1 recall (100%), and near-perfect (97-99%) Type-2 recall. The pattern-based configuration has strong VST3 (98%) and ST3 recall (92%), and matches NiCad for the top Type-3 performance of these tools. The token-based configuration has strong VST3 recall (92%), but loses some ST3 recall (65%), and has second best Type-3 recalls, just slightly better than SourcererCC (VST3: 92%, ST3: 63%). The line-based configuration has good VST3 recall (90%), but poorer ST3 recall (40%), falling just behind SimCad (VST3: 89%, ST3: 47%), but ahead of iClones (VST3: 30%, ST3: 5%). CloneWorks is the best performing tool, alongside NiCad and followed by SourcererCC. The other tools generally perform well for Type-1 and Type-2, but have poorer Type-3 performances. None of the tools have notable recall for the MT3 category.

12.5.3 Precision

We measured the precision of the clone detection tools by manually validating a sample of their detected clones from the BigCloneBench recall experiment. For each tool, we randomly selected 400 clone pairs for validation. This is a statistically significant sample size, with a confidence level of 95% and interval of 5%. Four clone experts, including one professional developer, were tasked with validating these clones as true or false positives. The clones from each tool were shuffled together, and the judges were kept unaware of which tool a given clone was reported by. The judges were asked to validate the clones as per their judgment, and were encouraged to mark as false positives the clones that were only coincidentally similar, or were of poor reporting quality (e.g., clones of only import statements, overlapping clones, etc).

The precision results are summarized in Table 12.2. CloneWorks (line) has perfect precision (100%), fol-

lowed by CloneWorks (token) with a precision of 95%, which compliments its competitive recall performance. CloneWorks (pattern) has lower but competitive precision at 86%, which compliments its top recall performance. SourcererCC has good precision, but its recall falls behind CloneWorks. CtCompare and SimCad have very poor precision. We find that CtCompare has poor reporting quality, while SimCad was reporting many dissimilar code fragments. SimCad has been shown to have good precision for standard software systems [135], so perhaps its SimHash algorithm is too sensitive for accurate inter-project detection.

The recall and precision of the three CloneWorks configurations implies a multi-pass detection strategy. The user can start with the line-based configuration, which will detect the most similar and highly relevant clones with perfect precision. They can then use the token-based configuration to extend their recall, although possibly with a few rare false positives. Then if high recall is very important to their use-case, they can further extend recall using the pattern-based configuration, however knowing that this may require removing some false positives from the extended results. We further measure the precision of CloneWorks for different user-guided features using 25K systems by manually validating more than 16K clone pairs in Section 12.5.6.

12.5.4 Execution Time and Scalability

Execution time primarily scales with the size of the input in terms of the number of lines of code (LOC) needing to be processed and searched for clones. We built test inputs for each order of magnitude ranging from 100KLOC to 250MLOC by randomly selecting source files from IJaDataset. The inputs were built in succession, such that each larger input is a superset to the smaller inputs, ensuring a progression in execution requirements, with only variation in LOC. The 250MLOC input is the full IJaDataset.

We want a clone detector that can scale with good execution time even on standard hardware, as these are the computers that are most accessible and affordable to researchers. For our average workstation, we use a computer with a 3.5GHz quad-core i7 CPU, 12GB of memory and an SSD running Ubuntu 16.04. We consider this a typical, affordable and accessible workstation.

The execution time and scalability results for the average workstation are summarized in Table 12.3. Scalability limits are marked by 'MEM', 'LIM' or 'ERR', where 'MEM' means the tool failed due to running out of available memory, 'LIM' means the tool failed due to internal input size limits built into the tool, such as fixed data structure size limits, and 'ERR' means some other error. Input partitioning with CloneWorks was only necessary for the full IJaDatsaet.

CloneWorks is able to scale to the full IJaDataset, and has the best execution time for the large inputs. SourcererCC also scales to 250MLOC, but with significantly longer execution time. CloneWorks (Type-3 token) reduces execution time by two order of magnitude, with comparable recall and precision to SourcererCC, while CloneWorks (Type-3 pattern) significantly improves Type-3 recall, while achieving an execution speed still one order of magnitude faster. The other tools reached design limits or out of memory conditions before the 250MLOC input. CCFinderX scaled to 100MLOC, but encountered unrecoverable parsing errors in the 250MLOC input.

Table 12.3: Scalability and Execution Time

Tool / LOC	100K	1M	10M	100M	250M
CloneWorks-T3(Line)	3s	28s	8m50s	1h27m	2h7m
CloneWorks-T3(Token)	4s	52s	6m19s	2h5m	4h1m
CloneWorks-T3(Pattern)	6s	59s	9m13s	2h43m	10h12m
CCFinderX	3s	30s	19m1s	6h51m	ERR
CtCompare	1s	11s	MEM	-	-
iClones	3s	23s	MEM	-	-
NiCad	20s	3m57s	1h38m	LIM	-
SimCad	31s	6m32s	1h33m	LIM	-
Simian	1s	6s	55s	MEM	-
SourcererCC	5s	21s	13m35s	19h41m	109h49m

For the largest inputs (100MLOC, 250MLOC), CloneWorks has the fastest execution time of the tools. For the 100MLOC input, it falls only behind Simian. While Simian is very fast, it only detects Type-1 and Type-2 clones, and despite this we find it has poor precision. For the smaller inputs (100KLOC, 1MLOC) CloneWorks is not the fastest, but has comparable execution time to the fastest tools. For these inputs, most of the tools complete within tens of seconds

Scalability and execution time become a problem with clone researchers target large inputs, such as IJaDataset. We have shown that CloneWorks has the best scalability and execution time in this case. Our input partitioning scheme means CloneWorks can continue to scale within memory to any input size. CloneWorks can use any number of available cores, so extraordinary hardware with many cores can be used to accelerate execution time even further.

12.5.5 Characterizing CloneWorks Performance

In the previous subsection, we empirically evaluated the performance of CloneWorks for our target: a large-scale inter-project source dataset on the order of hundreds of millions of lines of code. We found good execution time and scalability with CloneWorks. We expect similar performance for any inter-project source-code datasets built by crawling open-source repositories. Here we discuss the dominating complexities of CloneWorks' input converter and clone detector, and the factors that affect them.

Input Converter Complexity The input converter extracts the code fragments and then executes the code-fragment and term processors per code fragment before writing the resulting code fragments as term sets to a file. In terms of the size of the input, extraction scales with the number of source files, and code-fragment/term processing scales with the number of code fragments.

Code-fragment extraction and each of the code-fragment processors and term processors will have their own complexities over the size or properties of the source files and code fragments. However, these do not scale with the size of the input. It does not matter if a given source file is in a small subject system or a large inter-project dataset, it will have the same code-fragment extraction cost, and its code fragments will

have the same code-fragment and term processing costs. We can then consider these as constant costs in the complexity.

Therefore the complexity of the input converter is approximately $O(\alpha f + \beta n)$, where f is the number of source files, n is the number of code fragments, α is the amortized cost of code-fragment extraction per source file, and β is the amortized cost of the code-fragment and term processing per code-fragment.

The α cost will depend on the properties of the input, including the size of the source files and the complexity in building their ASTs for code-fragment extraction by sub-tree search and extraction. In general, this cost is very small for the average source file, perhaps just milliseconds on a single execution thread. Of course, an extraordinary source file, such as a very large file or a very dense/complex file in terms of AST construction, may require more significant time. These should be rare in most inputs, and will not significantly affect α. However, if these do become problematic, it is possible to tune the maximum stack size of the TXL-based parser to skip source files that require too much execution time to parse. This is primarily done to skip malformed source files that cause long parsing time before a syntax error is confirmed, but can also be used to improve execution time by skipping files that are expensive to parse.

The β cost will depend on the code-fragment processors and term processors used. Since the user can use any number of processors in any order, including their own custom processors by a plug-in architecture, β is strongly dependent on the configuration. When performing clone detection in large inter-project datasets, the user will want to prefer fast and inexpensive processing to keep β low. When performing clone detection in average software systems, a high β due to expensive processing may not cumbersome. Generally, the processing is going to scale in terms of the code-fragment size measured in lines, tokens or terms. If a given processor has good execution time but bad complexity, the input converter can be configured with a maximum code fragment size to keep β low at the cost of missing clones of the large code fragments.

In summary, the complexity of the input converter is linear in the number of source files and the code fragments they contain at the target granularity. Execution time depends on the average costs of code-fragment extraction and the code-fragment/term processing used. CloneWorks has configurations which can help prevent outliers source files and code-fragments from causing significant increases in the average extraction and processing costs. The actual runtime depends on the many configurations of the input converter, and the properties of the actual input under analysis.

Clone Detector Complexity The dominate factor in scalability of the clone detector is the comparison of the code fragments with the modified Jaccard similarity coefficient. Given an input with n code fragments, in the worst-case $\frac{n(n-1)}{2}$ pairs of code fragments must be compared. Each comparison requires worst-case $O(m)$ time, where m is the number of terms in the smaller code fragment. Therefore clone detection has a worst-case complexity of $O(mn^2)$, where in this case m is the average (amortized) size measured in terms of the smaller code fragment in the $O(n^2)$ code-fragment comparisons. However, a number of optimizations reduce the actual computation done

We reduce the number of code fragment pairs that need to be compared by (1) using sub-block filtering optimization, (2) skipping the comparison of overlapping code fragments, and (3) skipping the comparison of code fragments that are too different in size to satisfy the similarity threshold. In our empirical evaluation, we found these optimizations to be very effective.

When we execute clone detection on IJaDataset for function code fragments 10 lines in length or larger we find 4.7 million function code fragments, which results in 11 trillion unique code fragment pairs to evaluate with the similarity metric and threshold. Using the Type-3 (line) configuration with a threshold of 70%, our optimization reduces the number of comparisons to 150 million code fragment pairs, a reduction of five orders of magnitude, and detecting 114 million clone pairs. With the Type-3 (token) configuration and a 70% threshold, our optimization reduces the comparisons to 19 billion, a reduction of 3 orders of magnitude, and detecting 146 million clone pairs. With the Type-3 (pattern) configuration at 70% threshold, out optimization reduces the comparisons to 150 billion, a reduction of two orders of magnitude, and detecting 347 million clones. The optimization performs worse with the Type-3 (pattern) configuration as the significant normalizations decrease the range of unique terms and lower the chance that the prefix of a code fragment consists of only rare terms, therefore causing more code fragment pairs that don't meet the threshold to by missed by sub-block filtering. Therefore, the representation of the code fragments as terms as determined by the input converter configuration, and the actual terms that exist within a given input, determine the effectiveness of the sub-block filtering optimization and the actual performance of the algorithm. We find out three Type-3 configurations perform well in a real large-scale inter-project dataset.

We reduce the cost of comparing two code fragments with the modified Jaccard similarity metric by tracking an upper and lower bound on the similarity metric. A pair of code fragments can be accepted as a detected clone as soon as the lower bound meets or exceeds the threshold, or rejected as soon as the upper bound is falls below the similarity threshold. While measuring the similarity metric is worst-case $O(m)$ in the size of the smaller code fragment measured in terms, often a decision is made before all of the terms have been processed.

Therefore, the complexity of the clone detector is approximately $O(\frac{n^2}{\gamma}\frac{m}{\epsilon})$, where γ is a reduction factor due to the sub-block filtering optimization and other optimizations that let us skip code-fragment comparisons, and ϵ is a reduction factor given that we do not need to fully compute the similarity value of two code-fragments under comparison. Of course, these reduction factors will depend on the input software system or source datasets under analysis, the representation of the code fragments as term sets as determined by the input convert configuration, and the similarity threshold used.

Space Complexity The input converter does not require significant memory, and the memory requirements do not scale with the size of the input. Code fragment extraction memory requirements scale with the size and complexity of the source files, but the parser allows a stack limit to be imposed to prevent malformed or outliers source files from requiring significant space or time to parse. The included code-fragment and term

processors also do not require significant space for the typical code fragment, and we have not encountered any space issues when processing IJaDataset. Of course, users could plug-in custom processors which do require significant space requirements.

The clone detector has three significant memory requirements: (1) an index over the range of terms linking them to their global term frequencies, (2) an inverted index over the range of prefix terms linking them to the code fragments with them in their prefix, and (3) each code fragment as a set of terms and as a ordered list of terms (with a pre-computed sub-block prefix). The space requirements of the global term frequencies object scales with the number of unique terms in the input. The space requirements of the inverted index scales with the number of prefix terms across the code fragments (one insertion per prefix term), which in worst-case is linear in the total number of terms (if a threshold of 0%). Storing the code-fragments in memory as both sets and ordered lists for fast and optimized computation of the modified Jaccard similarity coefficient requires space linear in the total number of terms across all of the code fragments. The terms themselves must also be stored as strings, which are interred to avoid storing duplicate strings. The space requirement of this is linear in the total number of characters across the unique terms. In summary, the space requirements is linear in: (1) the number of code fragments, (2) the number of terms, and (3) the number of characters across the unique terms (interred strings).

The memory requirements of clone detection can become very significant for large inputs. However, our partitioning scheme (Section 12.3) allows us to linearly reduce the space requirements by partitioning the problem. In our experimental evaluation (Section 12.5.4), we find that only for the 250MLOC (full IJaDataset) did we need to use partitioning while limiting execution to 10GB of memory.

In summary, space complexity is not a major factor with CloneWorks. The input converter has approximately constant space complexity with respect to the size of the input software system or source dataset. The clone detection has linear space complexity with respect to size parameters of the input code fragments and their terms, but can be reduced to execute within standard memory limits of a personal workstation using partitioning.

12.5.6 User-Guided CloneWorks

We have shown above that CloneWorks is the state of the art tool for general clone detection. It has the best scalability and execution time for large inputs, and matches or exceeds the best of the competing tools in recall or precision performance for Type-1, Type-2 and Type-3 clones. Now we evaluate the user-guided aspect of CloneWorks.

In Section 12.4 we showed how the user has full control over how the source-code is transformed and split into terms for clone detection. Users can add source-level transformations as code fragment processors, choose how to split into terms, and add term-level transformations as term processors. Using the plug-in architecture, there is unlimited possibilities. With the user-guided input converter, the user can achieve any representation of their code fragments, in order to detect any type or kind of clone and pursue novel clone

detection studies and their applications thereof.

To evaluate the user-guided aspect, we discuss here some of the transformations made possible by CloneWorks, and the scenarios or tasks they would be useful for, including: identifier normalization, syntax abstraction, syntax filtering, term processing and term representation. All the customizations and processors discussed here are implemented in the current version of CloneWorks. Of course, the user can further extend these with the plug-in architecture.

We empirically test a few of these possibilities in case studies with IJaDataset [4], a large inter-project Java dataset with 25K systems. To show we also support other languages, we repeat a number of these case studies for a C (postgresql) and a C# (mono) system. The effects of the user-guided approach will depend on the system under study, so we focus on Java as our IJaDataset lets us see their average effect across 25K systems. We begin by detecting clones with our Type-3 (line) configuration as a base case. We then customize this configuration to target particular kinds of clones using the user-guided aspects. We measure how many additional clones we detect using the customizations, and specifically measure precision for these targeted clones, validating up to 1000 clones for each of the 22 measurements of precision. In total, 16,426 detected clones were validated by seven graduate students with knowledge of clones and clone research, which we believe is the largest evaluation of precision for any one clone detector. The validators were given the same instructions as our precision experiment in Section 12.5.3. The validated clones are publicly available for replication studies and comparison with other tools [1]. We find that the user-guided aspect allows us to target and detect many unique kinds of clones with high precision. This precision measurement is in addition to the precision experiment in Section 12.5.3. The results are shown in Table 12.4, and we discuss the case studies in detail in their respective sections below.

Pretty Printing

CloneWorks uses a TXL-based pretty-printing to layout the parsed code-fragments in a consistent manner. This aligns the code fragments for splitting by newline, so the term sets can be compared using simple language-independent string matching. The default pretty-printing when split by newline results in code statements as terms. The user can easily customize the pretty-printing as needed for their clone detection task by modifying the formatting annotations in the grammar files. This allows the user to customize their term definition. The finer the granularity of decomposition of statements across multiple lines, the finer the granularity of comparison after the code-fragment is split into terms by line, and the more emphasis placed on this statement in the similarity measurement.

For example, when doing function granularity clone detection, we generally do not put emphasis on the method signature, so we format it on a single line which becomes a single term. A user may find the method signature is important for their experiment, and modify the pretty-printing to split its components across multiple lines (as shown below), with each line becoming a distinct term. This would put a greater emphasis on the method signature during similarity measurement, and allow partial similarity when only certain

Table 12.4: Demonstration of User-Guided Approach

Configuration	System	100% Threshold		70% Threshold	
		# Clones	Precision	# Clones	Precision
Type-3 line (base)	IJaDataset (Java)	16,557,878	100%	113,964,667	100%
Abstract Identifiers and Literals	IJaDataset (Java)	+62,478,746	100%	+233,345,709	89%
Abstract Arguments	IJaDataset (Java)	+2,462,012	79%	+36,556,835	76%
Filter Generics, Modifiers, Annotations	IJaDataset (Java)	+855,200	100%	+419,800	89%
Remove Exception Handling (Try-Catch)	IJaDataset (Java)	+101,477	100%	+1,641,232	96%
Fingerprinting	IJaDataset (Java)	+65,921,281	94%	-	-
Cross-Project API Usage Clones	50 Systems (Java)	-	-	7685	99%
Type-3 line (base)	Postgresql (C)	1	100%	437	100%
Abstract Identifiers and Literals	Postgresql (C)	+184	98%	+1053	97%
Type-3 line (base)	Mono (C#)	9540	100%	194,020	99%
Abstract Identifiers and Literals	Mono (C#)	+95,626	100%	+452,784	97%
Filter Generics, Modifiers Annotations	Mono (C#)	+368	100%	+436	94%

components of the signature are modified after cloning. The user could do this for any syntax structure(s) they are particularly interested in for their clone study or task.

```
public static
int mymethod(
int one, int two
) throws MyException
```

As an additional example, considering the pretty-printing of the for loops below. Typically, the opening of a for statement is formatted on a single line. When splitting by line to produce terms, this would cause code-fragments with the same for-loop opener to share a single term. However, since for-loop openers contain three distinct parts, if we pretty-print these parts on separate lines, then we can represent the parts as distinct terms after splitting. Then two code-fragments with the same for-loop opener will share 3 terms instead of 1, which is proportional to the number of statements within a for-loop opener. This also makes the similarity metric more sensitive to localized changes. For example, the first two for-loop below are similar except for their initialization part. With specialized pretty-printing, they share 2 of 3 terms, instead of 0 of 1 term. We layout the for-loop opener to keep for-loop indicating syntax on each line so that these terms will only ever match terms from other for-loop openers. For example, `for (int i = 0` will only match the initialization part of a for-loop, and will not match with the statement `int i = 0;`. We make use of a number of clone-detection specific pretty-printing in our standard parsing.

```
for (int i = 0       for(int i = 1       for(line
; i < 10             ; i < 10            : lines) {
; i++) {             ; i++) {
```

Identifier Normalization

Essential for the detection of Type-2 clones is the normalization of identifier names. Differences in identifier names also occur in Type-3 clones, and normalization may be required to detect those Type-3 clones that appear too dissimilar at the token or line level before normalization to be detected as a clone. CloneWorks supports both consistent and arbitrary identifier normalizations. With consistent normalization, the first unique identifier in the code fragment, and all of its instances, is replaced with 'ID1', the second with 'ID2' and so on. This allows the detection of Type-2 clones with consistently renamed identifiers. With arbitrary normalization, every identifier is simply replaced with 'ID'. Arbitrary renaming is also needed for Type-3 detection, where consistent renaming is ineffectual as the identifiers may not align. Using term processors and simple token optimizations it is possible to normalize only identifiers of a certain type such as variable names, method names, type names or primitive types. Targeted normalization can help isolate Type-2 clones with specific kinds of differences that affect their refactorability, making it easier for the user to understand and manage their Type-2 clones.

In Table 12.4 we show clone detection results with blind identifier renaming for Java, C and C#. We also include literal value abstraction to match our Type-3 (pattern) configuration used earlier. This normalization greatly increases the number of clones detected. With IJaDataset and the 100% threshold we find an additional 62 million clone pairs above the base configuration, and with the 70% we find an additional 233 million clone pairs. With the 100% threshold, we extend the base detection from Type-1 to Type-2 clones. With the 70% threshold, we are finding the Type-3 clones that also contain Type-2 differences that dropped their measured similarity with the base configuration. With the 100% threshold, this normalization does not negatively affect precision, with precision holding at 99-100%. With the 70% threshold, we only see a drop in precision in the Java case, although it is still good at 89%. This could be due to detection in a large inter-project dataset, where there is a higher chance of coincidentally similar code between systems.

Abstraction

CloneWorks includes a TXL-based [27] code-fragment processor for the abstraction of any syntactic structure in the target language's grammar. Abstraction is the replacement of a syntactic entity with an abstract representation such that differences in that syntactic element are removed before clone detection. Abstraction can be applied at the token level (e.g., literals), for whole statements (e.g., throw statements), or for parts of statements (initialization part of a for-loop). The user simply specifies the names of the syntax elements to abstract, and the abstraction is applied globally. For example, abstraction of the conditional part of an if-statement causes both `if(i==10)` and `if(!buffer.isEmpty())` to be replaced by an identical abstract form `if(condition)`. Abstracting a whole statement, such as throw statements, replaces all such statements with its name: `throw_statement`. Users can easily apply advanced abstractions, for example only abstracting if conditions when the condition is a function call, by writing simple TXL rules.

Abstraction is needed to detect clones with specific kinds of edits. Applying abstractions and then

detecting clones with a 100% threshold can detect clones that differ by only a certain kind of difference. Abstractions could be combined to detect clones with specific combinations of differences. This allows us to target certain kinds of Type-3 clones even with a 100% similarity threshold. Some kinds of differences can cause Type-3 clones to have a low similarity measure. Lowering the threshold would detect these clones, but might also detect many false positives. Abstraction can be used to increase the similarity of Type-3 clones with particular kinds of differences above the threshold, without increasing the similarity of false positives with other kinds of differences. We discuss some example uses of abstraction below, including a case study.

Method Arguments Abstraction can be used to improve the recall of clones with particular kinds of differences. For example, code is often cloned, and an identifier method argument is replaced with an expression [9, 13]. Clone detectors rely on thresholds to detect these clones. CloneWorks can detect these even with a 100% similarity threshold by abstracting the method call arguments.

Loop Conditions A code fragment with for loop(s) may be copied and the looping conditions changed, possible with other modifications. We can detect these clones by abstracting the initialization, condition and afterthought parts of for-loops. With this abstraction, for-loops like `for(int i=0; i<10; i++)` are replaced by `for(init; condition; afterthought)`. Adding this to the Type-1 configuration will detect those clones only with changes in one or more of their for parts. Adding this to the Type-3 configurations will extend detection to those clones that fell below the similarity threshold due to differences in their for-parts. The user may not consider changes in for-loop parts to be significant in their system, so abstracting these parts is a good way to extend recall without dropping the similarity threshold, which could harm precision.

Whole Loops We could also abstract the whole loops, including its opening statements and contained code-block. With this abstraction, a loop like `for(int i=0; i<10; i++) {System.out.println(i);}` would be completely replaced by `for_loop_statement`. This allows the detection of clones that contain completely different loops. This may hurt precision as code-fragments that are mostly loops will become trivially similar. However, this may be necessary to detect certain clones in a software system. The user may decide the loss of precision is worthwhile for their system and use-case.

We can easily imagine some scenarios where this abstraction is necessary. Consider that a code-fragment has been duplicated that contains a loop. That loop is then extracted into a new function and replaced by a method call in one version but not the other. If the loop was a significant portion of the code-fragment, then this clone may no longer be detected even with a Type-3 configuration. Or consider the case where a code-fragment that accesses a database and queries a table is duplicated many times, but the loop that iterates through the query results is modified to perform a different computation. It is desirable to detect this clone as the database access code should be maintained together and possibly extracted and abstracted, but loops computing different calculations may be significantly different and cause the code-fragments to fall below reasonable Type-3 similarity thresholds. These clones could be detected by abstracting the for-loops.

Fingerprinting Abstraction can also be used for fingerprinting. If we abstract all simple statements (assignment, throws, declaration, etc) and the conditions of control statements (loops, if, switch, etc), we are left with the abstract structure of a code fragment (fingerprint). We can then detect clones with the same or similar programming structure and patterns.

This can have applications in bug detection by detecting clones of buggy programming patterns. While clone detection using fingerprints could have poor precision, this is less of an issue when we plan to only investigate the fingerprint clones of code fragments with known bugs or security flaws. An example of a code-fragment finger-print is shown in Figure 12.2

Case Study As a case study, we executed the abstract arguments and fingerprint cases for IJaDataset, and the results are shown in Table 12.4. There are 2.5 million clones in IJaDataset that only differ by their method call arguments, while 36.6 million are missed by the base configuration due to method arguments lowering their similarity below 70%. In addition to changes in identifier arguments, this also detects the cases where an identifier or expression argument is replaced with a different expression, which is a common occurrence in cloning [9]. However, argument abstraction has a hit on precision (76-79%). This is because in Java, developers sometimes place significant parts of their code within method call arguments, such as long call chains and anonymous classes.

With fingerprinting, we used only the 100% threshold as the 70% threshold is too relaxed with such significant abstractions. We find 66 million additional clones with this abstraction and the 100% threshold. With fingerprinting, we are finding the code fragments that have the same overall structure in terms of kinds of statements, but which may express these statements differently. With a 100% threshold we are finding clone pairs with the same programming pattern, and which are often Type-3 clones, and we are finding more clones with good precision at 94% despite heavy abstraction.

Filtering

CloneWorks includes a TXL-based [27] code-fragment processor that can filter any syntax element before clone detection. For example, filtering generics syntax would transform the statement `List<String> strings;` into `List strings;`. In this way, the user can detect clones ignoring certain syntax elements during similarity measurement. CloneWorks can filter parts of specific kinds of statements, whole kinds of statements, or even significant syntactical structures. Filtering is applied globally to all of the code-fragments. More advanced filtering, which might require conditional filtering or additional transformation, can be done by the user by implementing simple TXL rules.

With a 100% similarity, filtering will detect the clones that differ only by the removed syntax. With a lower threshold (e.g., 70%), filtering will allow the detection of clones that previously fell below the threshold due to dissimilarity caused by the targeted syntax. Filtering syntax that may cause dissimilarities unimportant to the user is an alternative to reducing the similarity threshold when trying to improve recall, and could

```
|=============================================================================================|
| ORIGINAL CODE FRAGMENT                                          | NORMALIZED CODE FRAGMENT  |
|=============================================================================================|
| private void fireTargetRemoved(TargetEvent targetEvent) {       | method_header {           |
|   TargetListener[] listeners = listenerList.getListenerList(); |   local_variable_declaration |
|   for (int i = listeners.length - 2; i >= 0; i -= 2) {          |   for (condition) {       |
|     try {                                                       |     try {                 |
|       if (listeners[i] == TargetListener.class) {               |       if(condition) {     |
|         listeners[i + 1].targetRemoved(targetEvent);            |         expression_statement |
|       }                                                         |       }                   |
|     } catch (RuntimeException e) {                              |     } catch (condition) { |
|       LOG.warn("An error was thrown.");\                        |       expression_statement |
|     }                                                           |     }                     |
|   }                                                             |   }                       |
| }                                                               | }                         |
|=============================================================================================|
```

Figure 12.2: Example of Fingerprint Abstraction

help preserve precision. As well, filtering allows the targeting of certain kinds of clones, and helps address common clone detection scenarios. We discuss a few examples of filtering below, including a case study.

Irrelevant Syntax Filtering can remove syntax elements that cause mismatch of otherwise similar source lines/terms during clone detection, even when those elements do not have much of an impact on whether two code fragments are clones. For example, access modifiers, generics and annotations should perhaps be filtered before clone detection, as they are minor changes in otherwise identical syntax.

Assertions Some developers make use of assertions during development and maintenance of the software. However, it may not be desirable for the assertions to be considered during clone detection. In particular, if assertions are not used everywhere in the code, and when differences in assertions may cause Type-3 clones to be further syntactically dissimilar. Therefore, developers that make use of assertions may want to filter the assertion statements before clone detection.

Declaration and Simple Initialization Statements Declaration and (simple) initialization statements can be detrimental to clone detection. Code fragments with long lists of variable declaration/initializations can be found to be trivially similar, especially when identifier normalization is used, leading to false positives. Conversely, differences in declaration and initializations can cause otherwise similar cloned code fragments to not be detected.

Filtering the declaration statements and simple initialization statements could allow additional clones to be detected, and some false positives to be eliminated. We write a custom filtering program in TXL to achieve this. This program filters the declaration statements (e.g. `int x;`) and simple initialization statements (e.g., `int x = 0` and `int x = y`) from the code-fragments. It does not filter the initialization statements that assign an expression (e.g. `int x = y + z`), as these contain important program logic, but

instead normalizes them to an assignment statement by filtering the declaration aspect (e.g. `x = y + z`).

When used with a 100% threshold, this filtering allows the detection of clones that are nearly identical except for differences in declaration and simple initialization. With a threshold such as 70%, filtering the declaration and simple initialization statements allows the detection of clones that otherwise fell below the similarity threshold (70%) due to differences in these often more trivial statements. This filtering may cause some clones to no longer be detected when their similarity is mostly within the declaration statements, would could be a benefit when trying to find the most relevant clones.

Exception Handling Differences in exception handling can cause otherwise identical or similar code fragments to appear different and be missed during clone detection. This can occur when a code fragment is cloned, and its exception handling evolves independently. For example, one version might simply throw its exceptions while the other logs and handles them. We developed an advanced filtering processor that removes exception handling by searching for instances of try-catch-finally blocks and replacing them with only the code within the try and finally blocks, effectively filtering the exception handling. We implemented this in TXL using replacement rules like the one shown in Figure 12.3. An example of a code-fragment after the application of this rule is shown in Figure 12.4.

Retain Statements Filtering can also be used to remove all except a particular kind of syntax for clone detection. For example, we can filter all statements except switch statements, retaining only the switch condition and cases, but not the surrounding statements, or the statements within the switch cases. This would detect code fragments with the same or similar switch skeletons as clones. Switch cases encode domain knowledge that must be properly maintained and evolved, and detecting these cloning relationships would help the developer synchronize the domain knowledge during evolution. This kind of clone detection has uses in program comprehension, such as linking code fragments with similar domain knowledge, enabling knowledge extraction from poorly documented code and design recovery from poorly structured code. This filtering would also be interesting for if-else chains and for-loop headers.

```
rule removeTryCatchFinally
    replace $ [statement]
        'try '{ TryBlock [repeat declaration_or_statement] '}
        CatchPart [repeat catch_clause]
        'finally '{ FinallyBlock [repeat declaration_or_statement] '}
    construct CombineStmts [repeat declaration_or_statement]
        TryBlock FinallyBlock
    by
        CombineStmts
end rule
```

Figure 12.3: TXL Code for Removing Exception Handling

```
| ORIGINAL CODE FRAGMENT                                  | NORMALIZED CODE FRAGMENT                          |
|---------------------------------------------------------|---------------------------------------------------|
| public void method(Type1 obj1, Type2 obj2) {            | public void method(Type1 obj1, Type2 obj2) {      |
|   try {                                                 |   obj1.method1(obj2);                             |
|     obj1.method1(obj2);                                 |   obj2.method2(obj1);                             |
|     obj2.method2(obj1);                                 |   obj1.dipose();                                  |
|   } catch (Exception e) {                               |   obj2.dipose();                                  |
|     logger.print(e + ": " + obj1 + ", " + obj2);        | }                                                 |
|   } finally {                                           |                                                   |
|     obj1.dispose();                                     |                                                   |
|     obj2.dispose();                                     |                                                   |
|   }                                                     |                                                   |
| }                                                       |                                                   |
```

Figure 12.4: Example of the Try-Catch-Finally Normalization

Case Study As a case study, we executed our trivial syntax filtering example (generics, modifiers, annotations) for Java and C#, with the results shown in Table 12.4. We find that many clones are missed both with the 70% and 100% threshold due to these trivial (from the perspective of cloning) syntax differences between code fragments. For IJaDataset we find an additional 855,200 clone pairs with the 100% threshold, and an additional 419,800 clone pairs with the 70% threshold. Generally we find that this filtering does not harm precision (94-100%), with only a noticeable reduction in precision (89%) in the Java case with a 70% threshold. We also executed our exception handling filtering example for IJaDataset, with the results shown in Table 12.4. We find 101,477 clones that are identical (100% threshold) only when we ignore exception handling, showing that developers copy and paste code and then modify only the exception handling or evolve it differently. With the 70% threshold, we find that 1.6 million clones were not found by the base configuration due to differences in their exception handling. We find good precision with filtering exception handling (96-100%).

Term Processing

After the code-fragments have been pretty-printed, normalized, abstracted, etc. and then split into lines or tokens, CloneWorks allows flexible processing of the terms before clone detection. As discussed previously in Section 12.4, the term processors receive the terms in their original order, and output the same after processing. Term processors must be implemented as Java classes which can be plugged into CloneWorks, although they are free to execute external processes to help their computations. CloneWorks includes a number of useful term processors.

A number of term processors are designed for the case where the code-fragments are split into language tokens. `FilterSeperators` and `FilterOperators` remove specific tokens that might hurt the precision of clone detection across sets of language tokens. These can be used as the basis to build processors for filtering any token types. `NormalizeStrings` can be used to normalize string tokens to a common value,

`SplitStrings` splits string tokens into their words, while `Stemmer` stems identifiers which might help in the case of slightly renamed identifiers (e.g., `Tokenize` and `Tokenizer` are stemmed to `Tokeniz`).

A number of term processors are designed for both split by line and split by token. The `Joiner` processor combines all the terms into a single term with uniform whitespace delimitation, which can be used for exact (after normalization) detection. We have a `Hashing` processor which replaces the term strings with a hash string. This can avoid long string comparisons and reduce memory requirements when term strings are long, for example when the `Joiner` processor is used. `Hashing` supports both MD5 and SHA hashing, so collisions should be very rare. As well we have the `NGram` processor, which applies an n-gram filter across the terms. This can be used to maintain some token/line order information in the term list after they are converted to an unordered set for similarity measurement, although this can reduce recall when Type-3 edits are dispersed. We have a `RetainUnique` processor which removes duplicate terms when it is desirable to ignore term frequency during detection.

Of course, the user can easily implement any term processing they need and plug it into CloneWorks.

Custom Term Representation

With the input converter, any definition of a term, or representation as term sets, is possible. By customizing the pretty-printing, applying transformations with code-fragment processors, and then splitting by line, the developer can produce any representation imaginable. By customizing the representation of a code fragment as terms, for any definition of a term, the user can explore new clone types, which could have many applications across software analysis research.

For example, we could represent the code fragments by the set of normalized API calls they contain. We use a code fragment processor to transform the code fragments into a newline delimited list of normalized API calls they contain, in the order they are called. We built such a processor using SrcML [24], and an example transformed code fragment is shown in Figure 12.5. Splitting by line results in our target of code fragments as sets of normalized API calls for clone detection. This representation would allow us to detect clones of API usages. This would be useful to researchers studying Internet-scale code duplications, as developers learn API usage patterns from the web. It could also be used by API designers to see how open-source is using their APIs.

Term processors could then further refine or customize this detection. An NGram term processor would limit the detection to code fragments with similar API call chains. Alternatively, duplicate terms could be removed by `RetainUnique`, and API calls are used as a set of topics about a code fragment. Detecting code fragments with similar topics or that perform similar actions could be useful for design recovery, reverse engineering and restructuring [86, 118].

As a case study, we executed our API usage clone detection across 50 open-source Java systems taken from GitHub and SourceForge. We used our API term processor to transform the code-fragments into their list of API calls and then split by line. We used our `RetainUnique` term processor to remove duplicate

Figure 12.5: API Call Extraction Example

```
=============================================================================
| ORIGINAL CODE FRAGMENT                                    | TRANSFORMED     |
=============================================================================
| public void endOverlay() {                                | requestFocus(0) |
|  fContainer.requestFocus();                               | setVisible(1)   |
|  if (fEditScrollContainer != null) {                      | remove(1)       |
|   fEditScrollContainer.setVisible(false);                 | getBounds(0)    |
|   fContainer.remove(fEditScrollContainer);                | getX(0)         |
|   Rectangle bounds = fEditScrollContainer.getBounds();    | getY(0)         |
|   fContainer.repaint(bounds.getX(), bounds.get(Y),        | getWidth(0)     |
|               bounds.getWidth(), bounds.getHeight));      | getHeight(0)    |
|  }                                                        | repaint(4)      |
| }                                                         |                 |
=============================================================================
```

API call terms. This allows us to represent the code fragments as the set of API topics they contain. We performed clone detection with a 70% threshold and considering the code-fragments with at least 5 unique API terms (to consider only those implementing an API usage). We targeted inter-project detection, to find API usage duplication between projects, and found 7685 API clones. We validated 1000 of these clones and found a precision of 99%. These are not traditional clones, and often the code fragments are syntactically dissimilar, but use the same APIs to perform a common task. These kinds of clones may be useful to those studying the propagation of API usage patterns between software projects. Our goal here was not to perfect API clone detection, but with this example we have demonstrated how the user-guided approach can be used to target new kinds of clones for studies in software engineering practices.

12.6 Limitations

A limitation in all clone studies is tool configuration [139]. We configured the tools by evaluating permutations of their settings against BigCloneBench, and choosing settings that appeared to maximize recall while balancing precision. We could not exhaustively evaluate every configuration permutation, and we may not have found the best configurations. However, since BigCloneBench contains a wide variety of clones, we are confident we found good general configurations for accurate comparison of tool performance.

CloneWorks relies on the sub-block filtering optimization [116] for scalability in execution time. The performance of this optimization depends on the distribution of the code terms of the input code fragments, which depends both on the properties of the input source system and on the configuration of the input converter. The optimization will work best when there is a variety of terms, both common and rare, and with the code fragments containing a mix of both. In this study, we demonstrated good scalability across a large variety of software systems (IJaDataset), and with a variety of input converter configurations. When configuring the input converter, the user should take some care to ensure their normalization and transfor-

mations preserve unique aspects about the code-fragments such that the sub-block filtering optimization is successful.

12.7 Related Work

Rattan et al. [104] found at least 70 clone detectors in the literature. However, very few scale to large inter-project repositories like IJaDataset, and they have limited user-guided features. There have been CCFinderX [58], CtCompare [133], iClones [41], NiCad [110], SimCad [136], Simian [44], and SourcererCC [116], which we extensively compared with CloneWorks in Section 12.5.

Liveri et al. [84] scaled CCFinder [58] using input partitioning. The large input was partitioned into smaller inputs, and CCFinder was executed for each pair of partitions. Scalability in execution time was achieved using a large compute cluster. Ishihara et al. [48] scale the detection of Type-1 and Type-2 method clones by comparing their MD5 hash values after normalization. However, this does not detect the important Type-3 clones. Hummel et al. [47] proposed the use of indexes for scalable clone detection in large inter-project repositories. However, their index is quite large, requiring the index and computation to be distributed over a compute cluster, and their technique detects only Type-1 and Type-2 clones. Others have scaled clone detection in domain-specific ways, which cannot be used for general detection. Koschke [73] scaled license violation detection using suffix trees. Chen et al. [22] detect Android application clones in application marketplaces. Keivanloo et al. [61] use a clone index to scale code search to large inter-project repositories. Our [126] Shuffling Framework scales existing clone detectors, without modification, to large repositories using non-deterministic input partitioning with file pair shuffling and filtering optimizations. This scales the tools with some loss of recall, and requires a small cluster for scalability in execution time. The sub-block filtering approach was proposed by Sanjani et al. [116].

CloneWorks provides a finer granularity of control over the source transformations and processing, including a plug-in architecture for providing custom source transformation and processing. We provide a implementation and architecture of the sub-block filtering and partial index approach that uses fast, low complexity, but memory intensive, parallel data structures. Scalability within typical memory constraints is achieved using our novel input partitioning approach, which is inspired by the Shuffling Framework [126]. CloneWorks achieves top recall with good precision while reducing execution time by one or two orders of magnitude for IJaDataset compared to SourcererCC.

12.8 Contributions of CloneWorks

CloneWorks was built by combining and improving upon the best techniques found in the literature. These influencing works were acknowledged in this chapter and described alongside the other related work in the previous section. To better highlight the contributions of CloneWorks, we summarize in this section the

techniques which have been taken from the literature, how they were adapted or improved for CloneWorks, and what unique contributions we provide with CloneWorks.

CloneWorks measures clone similarity using a Jaccard-based metric. The use of Jaccard and similar set-based metrics is common in clone detection [63, 115, 116, 126]. We use this metric as it is efficient to compute (scalability) and simple to understand (essential to predict how source normalization/transformation will affect detected clones). A contribution of this work is we showed that the Jaccard-based metric can achieve high recall and precision in the detection of various types and kind of clones, including with various source-code normalizations and representations.

To achieve scalability in execution time, CloneWorks uses the sub-block filtering optimization with a partial clone index. The sub-block filtering optimization was introduced by Sajnani et al. [115, 116], and they proposed the use of a (partial) clone index to efficiently apply sub-block filtering to an input set of code fragments. The use of clone indexes to scale clone detection is a well known approach [47, 61, 115, 116, 126]. The first implementation of the sub-block filtering and partial clone index approach is in SourcererCC, which we introduced with Sajnani et al. [116]. We previously demonstrated that this approach provides a significant improvement in scalability compared to the other state of the art tools [116]. Our contribution with CloneWorks is a new implementation of this approach. Our multi-threaded implementation stores the index and code fragments in-memory for instant lookup and in data-structures optimized for efficient computation of the Jaccard metric and sub-block filter. We demonstrate that this improves execution speed by up to one to two orders of magnitude compared to the original implementation in SourcererCC.

Our new implementation is fast but has high memory requirements. To scale within the available memory of even a modest workstation, we use a novel deterministic input partitioning procedure designed specifically for the sub-block filtering and partial clone indexing approach. While input partitioning has previously been used for clone detection, including a general deterministic approach by Livieri et al. [84], and a general non-deterministic approach with our Shuffling Framework [126]; our contribution with CloneWorks is a custom deterministic approach specifically for index-based clone detectors.

The input converter is one of the primary contributions of CloneWorks. It is inspired by the flexible source normalizations options provided by NiCad [110]. CloneWorks extends this concept by allowing the user to fully customize the entire pipeline of the source-code parsing, extraction, normalization, transformation and representation. The input converter is designed for scalable clone detection with an optimized multi-threaded implementation. While NiCad allows the user to provide custom transformations via TXL scripts, CloneWorks allows custom transformations to be implemented in any technology.

12.9 Conclusion

In this chapter, we introduced CloneWorks, our fast, scalable and user-guided clone detector. It includes our user-guided input converter, which allows the user to fully customize their source code transformations

and representation for general-purpose and targeted clone detection experiments. Fast clone detection is achieved by our efficient fully in-memory Jaccard-based clone detector, which uses sub-block filtering with a clone index to scale in computation time, and input partitioning to scale within the memory constraints of a personal workstation. We compared CloneWorks against eight competing clone detection tools in an extensive evaluation experiment measuring and comparing their recall, precision, scalability and execution time. CloneWorks can scale to IJaDataset, a large-scale source-code repository containing 250MLOC, in just 2-10 hours with excellent recall and precision. To the best of our knowledge, CloneWorks is the fastest clone detector for large inter-project repositories. We demonstrated user-guided clone detection in a series of scenarios and case studies on a large inter-project dataset (25K systems) that demonstrate CloneWorks's strength in customized clone detection. The experimental dataset is available online [1] for replication and/or comparison purpose.

Part IV

Closing

Chapter 13

Conclusion

In this thesis, we advanced the state of art in clone detection tool evaluation and large-scale clone detection.

To advance the state of tool evaluation, we introduced two new clone benchmarks: the Mutation and Injection Framework, a synthetic benchmark that can measure recall at a fine granularity, and BigCloneBench, a real-world and large-scale clone benchmark that can measure recall for inter-project and intra-project clones of the four primary clone types, including across the entire spectrum of syntactical similarity. We used these benchmarks in a number of studies evaluating the state of the art tools as well as the benchmarks and evaluation procedures themselves. We compared our benchmarks against the previous and related work, and found ours are the highest quality benchmarks. Synthetic and real-world benchmarking is very complimentary, and provides a full understanding of clone detection recall.

To advance the state of large-scale clone detection, we introduced the Shuffling Framework for scaling existing clone detectors to large scale, and a dedicated tool, CloneWorks, for fast, scalable and user-guided large-scale clone detection experiments. The Shuffling Framework scales existing tools by reducing a large source-code input into a series of manageable subsets within scalability limits. It can successfully scale ordinarily non-scalable tools, but with a reduction to clone detection tool's native recall performance, and requiring a small compute cluster. With CloneWorks, we introduce a clone detection tool designed for achieving large-scale clone detection experiments, particularly in large inter-project source-code datasets. Compared to the state of the art, CloneWorks has the best execution time and scalability for large inputs. It uses an efficient and parallel architecture, which makes use of partitioning heuristics from our Shuffling Framework, and scalability heuristics from our related work [116]. CloneWorks has user-guided source-code parsing and transformation, which enables users to target any type or kind of clone. CloneWorks is the best tool for exploratory clone detection in large inter-project datasets.

The remainder of this chapter is organized as follows. Section 13.1 summarizes the contents of this thesis. Section 13.2 summarizes our contributions to the state of the art in clone detection tool benchmarking and evaluation, and large-scale clone detection techniques. In Section 13.3 we discuss directions of future research made possible by this thesis.

13.1 Research Summary

In Part I we introduced a synthetic clone benchmark, the Mutation and Injection Framework, which synthesizes corpora of reference clones using clone-producing mutation operators (Chapter 3). The advantage of this benchmark is it can measure the recall of clone detection tools at a fine granularity, not only per clone type but also for each kind of edit developers make on copy and pasted code of the first three clone types. The framework implements an evaluation procedure that allows recall to be compared per clone type and per edit type without bias. We demonstrated the Mutation Framework in a tool comparison experiment (Chapter 4) where we compared its results against our expectations and against a previous real-world clone benchmark, Bellon's Benchmark. We found that the Mutation Framework is accurate and had no anomalies in its results. In contrast, we found Bellon's Benchmark to be inaccurate for modern clone detection, with anomalies in its measurements, suggesting that a new real-world benchmark was needed (which we address in Part II). We measured the recall of the clone detection tools at a fine granularity for both block-level and function-level clones in Java, C and C# languages (Chapter 5). We also showed how the Mutation Framework can be extended for rigorous evaluation of any kind of clones, such as gapped clones (Chapter 6). We adapted our clone-producing mutation technology to create ForkSim, a framework for generating artificial software variants (i.e., forks) with known similarities and differences. ForkSim can be used to evaluate the recall of software variant analysis tools (including clone detectors) that are used to migrate software variants towards a software product line architecture. ForkSim is a demonstration on how our Mutation Framework technology can be adapted to create benchmarks in related software analysis fields.

In Part II we introduced a real-world benchmark, BigCloneBench (Chapter 9), which is a collection of reference clones in IJaDataset, a big inter-project source dataset (25K projects, 250MLOC). We built this benchmark by mining IJaDataset for clones of 48 distinct functionalities. We developed a novel clone mining approach which minimizes the manual clone validation efforts while also reducing subjectivity in the resulting clone benchmark. The advantage of BigCloneBench is its size and breadth of scope. It can be used to evaluate clone detection tools for all four primary clone types, for intra-project and inter-project clones, for semantic clones, and for clones across the entire spectrum of syntactical similarity. We used BigCloneBench in a tool comparison study (Chapter 9) and compared its results against our Mutation and Injection Framework, demonstrating the need for both real-world and synthetic clone benchmarks to fully understand clone detection tool recall. We implemented our procedure as a benchmarking framework called BigCloneEval (Chapter 10), which makes it easier for the community to replicate, extend and customize our study and evaluate their new clone detection tools using BigCloneBench.

In Part III we presented our work on large-scale clone detection. In Chapter 11, we presented our Shuffling Framework with which we scaled classical (ordinarily non-scalable) clone detection to large scale by reducing the large input into a series of smaller subsets. We explored the use of various heuristics for choosing these subsets. We found good results by building subsets of similar source files (identified by an inverted clone

index) and by tracking the pairs of similar source files already seen by the clone detector to avoid repetition. Then in Chapter 12, we presented our new clone detection tool, CloneWorks, which provides best-in-class scalability and execution time for inputs up to large inter-project datasets. CloneWorks uses a Jaccard-based similarity metric, and achieves scalability in execution time using the sub-block filtering heuristic with clone indexing, and scalability within commodity memory using an index-based input partitioning. CloneWorks is user-guided, allowing the user to customize the source transformations and normalizations applied before detection, to target any type of kind of clone. We performed a tool comparison study that extensively compared our CloneWorks against the competing tools in terms of recall, precision, execution time and scalability. We evaluated the user-guided aspect by customizing CloneWorks for various scenarios. As part of our case studies, we validated over 15K user-guided clones detected by CloneWorks. To our knowledge, this is the most extensive evaluation of precision for any one clone detection tool.

13.2 Contributions

Our research towards clone detection tool evaluation and comparison contributes to the state of the art in the following ways:

- **A synthetic clone benchmark: The Mutation and Injection Framework.** The benchmark can measure recall at a very fine granularity: for each type of edit developers make on copy and pasted code of the first three clone types. The implementation of the framework automates recall measurement experiment, including: clone synthesis, executing the clone detectors, recall measurement, and evaluation summary. The framework allows experiments to be shared, repeated, and extended. The framework can be extended to evaluate tools for any kind of clone using custom clone-producing mutation operators.

- **A software variant analysis benchmark: ForkSim.** ForkSim generates datasets of artificial software variants (i.e., artificial forks), with known similarities and differences. These can be used to evaluate software variant analysis tools, including clone detection tools, for tasks such as fork consolidation and migration towards a software product line. ForkSim is built upon our Mutation and Injection Framework technology, and demonstrates how this benchmarking strategy can be transfered to other software analysis domains.

- **A real-world and big clone benchmark: BigCloneBench.** BigCloneBench contains eight million reference clones across 48 distinct functionalities, including the four primary clone types, semantic and syntactic clones, inter-project and intra-project clones, and clones spanning the entire spectrum of syntactical similarity. It is the only clone benchmark for evaluating clone detection tools for large inter-project datasets, which has many potential applications. We also provide a benchmarking framework, BigCloneEval, that automates configurable recall measurement experiments on top of BigCloneBench.

- **Tool comparison studies evaluating and comparing the recall of modern clone detection tools, and the state of modern clone benchmarks.** We measure and compare recall using our Mutation and Injection Framework and the popular real-world benchmark Bellon's Benchmark. We show that Bellon's Benchmark is not appropriate for modern clone detection tools, and demonstrate the accuracy of synthetic benchmarking strategies. We then compare the Mutation and Injection Framework and our modern real-world benchmark BigCloneBench to demonstrate the need for both benchmarking strategies to get a complete understanding of the tools. We investigate recall for inter-project vs intra-project clones, for the spectrum of syntactical similarity, and evaluate how precisely the individual tools capture the reference clones. We also measure the precision, execution time and scalability of the state of the art tools, including our CloneWorks. We provide the most up to date and complete comparison studies of the state of the art tools.

- **Tool comparison studies evaluating recall at a fine granularity.** Using our Mutation and Injection Framework, we measure recall per clone type and per edit type (from the editing taxonomy for cloning) for block and function clones in Java, C and C# programming code. Additionally, as a demonstration of the extensibility of the Mutation Framework, we design mutation operators for producing gapped clones and measure the recall clone detection tools for different Type-3 gap lengths.

- **A tool comparison study evaluating clone detection tools for large-scale clone detection.** As part of our CloneWorks tool, we evaluated the recall, precision, execution time and scalability of the state of the art clone detection tools for large-scale clone detection.

Our research into large-scale clone detection contributes to the state of the art in the following ways:

- **Large-Scale clone detection using the classical clone detectors: the Shuffling Framework.** We investigated the use of non-deterministic input partitioning, coarse similarity analysis and heuristics to scale classical (natively non-scalable) clone detection tools to large inter-project source-code datasets. We found success by executing the clone detectors for a series of subsets of the large input, within their scalability limits, built by and randomly selecting pairs of similar source files without repetition. This approach exploits an inverted clone index with efficient tracking of the pairs of similar source files already exposed to the clone detector in previous subsets to avoid needless repetition. Our study found that we could successfully scale the classical detectors using a small number of commodity workstations at the cost of an acceptable loss in native recall performance.

- **CloneWorks: Fast, Scalable and User-Guided Clone Detection for Large-Scale**. CloneWorks is the best-in-class tool for execution time and scalability with large inter-project source datasets. Its recall and precision performances meets or exceeds the best of the state of the art tools. It is also the only large-scale clone detector to be user-guided, meaning the user can configure and customize it to target any type or kind of clones. In particular, it is the only scalable tool to have a plug-in architecture

for extensibility. For CloneWorks, we performed the most extensive tool evaluation and comparison study available in the literature.

13.3 Publications from this Thesis Research

Here we summarize our publications from and related to this thesis. In total we have published thirteen papers, including two journal papers and eleven conference and workshop publications.

Refereed Journal Publications

1. **Jeffrey Svajlenko**, Iman Keivanloo, Chanchal Roy, "Big Data Clone Detection Using Classical Detectors: An Exploratory Study," Journal of Software: Evolution and Process, vol. 27, no. 6, pp. 430-464, June, 2015. [**Special Issue Invitation**]

2. **Jeffrey Svajlenko**, Chanchal K. Roy, "A Machine Learning Based Approach for Evaluating Clone Detection Tools for a Generalized and Accurate Precision.", International Journal of Software Engineering and Knowledge. 32 pp. [**Special Issue Invitation**]

Refereed Conference and Workshop Publications

1. **Jeffrey Svajlenko**, Chanchal K. Roy, "Fast and Flexible Large-Scale Clone Detection with CloneWorks", In Proceedings of the Tool Demonstration Track of the 39th International Conference on Software Engineering (ICSE 2017), 4 pp., Buenos Aires, Argentina, May 2017. [32% Acceptance Rate]

2. **Jeffrey Svajlenko**, Chanchal K. Roy, "CloneWorks: A Fast and Flexible Large-Scale Near-Miss Clone Detection Tool", In Proceedings of the Poster Track of the 39th International Conference on Software Engineering (ICSE 2017), 2 pp., Buenos Aires, Argentina, May 2017. [Invitation]

3. **Jeffrey Svajlenko**, Chanchal K. Roy, "BigCloneEval: A Clone Detection Tool Evaluation Framework with BigCloneBench", In the Tool Demonstration Track of the 32nd International Conference on Software Maintenance and Evolution (ICSME 2016), Raleigh, North Carolina, October 2016.

4. **Jeffrey Svajlenko**, Chanchal K. Roy, "Efficiently Measuring an Accurate and Generalized Clone Detection Precision using Clone Clustering", In Proceedings of the 28th International Conference on Software Engineering and Knowledge Engineering (SEKE 2016), 426-433, Redwood City, California, July 2016. [**30% acceptance rate**] [**First Place Best Papers Award**]

5. Hitesh Sajnani, Vaibhav Saini, **Jeffrey Svajlenko**, Chanchal K. Roy, and Cristina V. Lopes, "SourcererCC: Scaling Code Clone Detection to Big Code", In Proceedings of the 38th International Conference on Software Engineering (ICSE 2016), 12 pp., Austin, Texas, May 2016. [**19% Acceptance Rate**]

6. **Jeffrey Svajlenko** and Chanchal K. Roy, "Evaluating Clone Detection Tools with BigCloneBench", In Proceedings of the 31st International Conference on Software Maintenance and Evolution (ICSME 2015), 10 pp., Bremen, Germany, September 2015. [**22% Acceptance Rate**]

7. **Jeffrey Svajlenko** and Chanchal K. Roy, "Evaluating Modern Clone Detection Tools", In Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME 2014), Victoria, Canada, September 2014, pp. 321-330. [**19% Acceptance Rate**]

8. **Jeffrey Svajlenko**, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy and Mohammad Mamun Mia, "Towards a Big Data Curated Benchmark of Inter-Project Code Clones", In Proceedings of the Early Research Achivements track of the 30th International Conference on Software Maintenance and Evolution (ICSME 2014), Victoria, Canada, September 2014, pp. 476-480. [**36% Acceptance Rate**]

9. **Jeffrey Svajlenko**, Chanchal K. Roy and Slawomir Duszynski, "ForkSim: Generating Software Forks for Evaluating Cross-Project Similarity Analysis Tools", In Proceedings of the Tool Paper track of the 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2013), Eindhoven, the Netherlands, September 2013, pp. 37-42.

10. **Jeffrey Svajlenko**, Chanchal Roy, and James Cordy, "A Mutation Analysis Based Benchmarking Framework for Clone Detectors", In Proceedings of Short/Tool Papers Track of the ICSE 7th International Workshop on Software Clones (IWSC 2013), San Francisco, CA, May 2013, pp. 8-9.

11. **Jeffrey Svajlenko**, Iman Keivanloo, and Chanchal K. Roy, "Scaling Classical Clone Detection Tools for Ultra-Large Datasets: An Exploratory Study", In Proceedings of the ICSE 7th International Workshop on Software Clones (IWSC 2013), San Francisco, CA, May 2013, pp. 16-22.

13.4 Future Research Directions

Expansion of BigCloneBench

As part of this thesis, we built BigCloneBench to include clones of 48 distinct functionalities, and found eight million reference clones. A portion of these clones were validated by multiple judges, so that we could measure the accuracy of our clone-mining procedure, which we found to be excellent. An avenue of future research would be to continue to expand BigCloneBench to additional functionalities. As well, additional validation work would continue to improve the benchmark. An interesting extension would be to move the clone mining and validation procedure to the cloud for community-wide collaborative efforts. As well, BigCloneBench could be expanded to include multiple reference corpora built on top of IJaDataset. We showed that we have taken the initial steps to adding a validated corpora built on tool-detection results. As well the Mutation Framework could be adapted to introduce synthetic clones to IJaDataset.

Higher-Order Mutations for Clone Synthesis and Benchmarking

With the Mutation and Injection Framework, we evaluated the tools using simple order-one clone-producing mutations. Complex clones could be synthesized using higher-order mutations (various combinations of order-one mutation operators). However, higher-order mutations are known to be unpredictable and could result in synthetic clones that are too different from real clones. An interesting future work would be to study higher-order clone-producing mutations, and to see whether they could be properly controlled to produce complex and realistic clones. For example, such a study could examine real clones in practice, and use properties about these clones to guide complex clone synthesis by higher-order mutations.

Domain-Specific Clone Benchmarks

In this thesis, we introduced two general benchmarks for measuring clone detection recall, one a using synthetic benchmarking strategy and one using real-world data. Also of interest are benchmarks for evaluating clone detection tools in a domain-specific ways. For example, a benchmark of only bug-related clones, or a benchmark of clones suitable for refactoring by function merging. We show an example of such domain-specific clone benchmark with ForkSim in this thesis. An interesting future work would be to produce more domain-specific clone benchmarks with the Mutation Framework principals. Of course, the greater the number of variety of clone benchmarks, the more confident we can be in the tool evaluation results, so new high-quality benchmarks are always valuable.

Improving the Measure of Clone Detection Precision

In our published work, we performed an exploratory study on using clone clustering to improve the measurement of precision. We found that choosing clones from a clustering yielded a wider variety of clones, and reduced biases in the measurement of precision by up to an order of magnitude. A future work would be to further explore this technique to standardize it as the method of measuring clone detection precision.

Exploratory Clone Detection in Large Inter-Project Datasets

With CloneWorks, we have produced a user-guided tool for large-scale clone detection experiments. A future work is to develop various configurations for the CloneWorks input builder in order to explore and discover new kinds of clones in large inter-project datasets. Our chapter on CloneWorks has motivated such studies, for example we showed how CloneWorks can be used to detect API usage clones, which could be used in an interesting exploratory study.

Source Transformation and Clone Detection

With CloneWorks, we can insert any kind of source transformation into clone detection, and produce different representations of the code fragments as terms sets affects clone detection results. CloneWorks makes it

possible to study how various source transformations, normalizations and alternate representations affects clone detection. Our benchmarks would allow an investigation in how various transformations affect recall and precision. There is great potential with CloneWorks to experiment within clone detection, as well to pursue new kinds of clones.

Automatic Clone Validation

Manual clone validation was frequently needed in this thesis, in particular for measuring precision and for building BigCloneBench. To overcome this effort-intensive task, an automatic or at least computer-assisted clone validator is needed. As part of this thesis, sufficient clone reference data has been produced to be used as an input for a machine-learning based approach for clone validation. The efforts required in this thesis and in the clone literature is motivation for such a work.

References

[1] Bigclonebench experimental artifacts, 2017. https://goo.gl/h8V1zw.

[2] R. Al-Ekram, C. Kapser, R. Holt, and M. Godfrey. Cloning by accident: an empirical study of source code cloning across software systems. In *2005 International Symposium on Empirical Software Engineering, 2005.*, pages 10 pp.–, Nov 2005.

[3] M. H. Alalfi, J. R. Cordy, and T. R. Dean. Analysis and clustering of model clones: An automotive industrial experience. In *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 375–378, Feb 2014.

[4] Ambient Software Evoluton Group. SECold IJaDataset 2.0. `http://secold.org/projects/seclone`, January 2013.

[5] Amplab. Big data benchmark. `https://amplab.cs.berkeley.edu/benchmark/`.

[6] Apache. Apache pig project. `http://pig.apache.org/`.

[7] Atlassian. Pigmix benchmark. `https://cwiki.apache.org/confluence/display/PIG/PigMix`.

[8] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of 2nd Working Conference on Reverse Engineering*, pages 86–95, Jul 1995.

[9] B.S. Baker. Finding clones with dup: Analysis of an experiment. *IEEE Transactions on Software Engineering*, 33(9):608–621, 2007.

[10] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377, Nov 1998.

[11] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377, Nov 1998.

[12] Ira D. Baxter, Michael Conradt, James R. Cordy, and Rainer Koschke. Software clone management towards industrial application (dagstuhl seminar 12071). *Dagstuhl Reports*, 2(2):21–57, 2012.

[13] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, Sept 2007.

[14] Stefan Bellon. Stefan bellon's clone detector benchmark. `http://www.softwareclones.org/research-data.php`.

[15] Stephen Bellon. Vergleich von Techniken zur Erkennung duplizierten Quellcodes. Master's thesis, Universität Stuttgart, 2002. 156 pp.

[16] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wasowski. A survey of variability modeling in industrial practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 7:1–7:8, New York, NY, USA, 2013. ACM.

[17] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwe. On the use of clone detection for identifying crosscutting concern code. *IEEE Transactions on Software Engineering*, 31(10):804–818, Oct 2005.

[18] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation*, pages 36–43, 2002.

[19] Alan Charpentier, Jean-Rémy Falleri, David Lo, and Laurent Réveillère. An empirical assessment of bellon's clone benchmark. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, EASE '15, pages 20:1–20:10, New York, NY, USA, 2015. ACM.

[20] Alan Charpentier, Jean-Rémy Falleri, David Lo, and Laurent Réveillère. An empirical assessment of bellon's clone benchmark. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, EASE '15, pages 20:1–20:10, New York, NY, USA, 2015. ACM.

[21] Alan Charpentier, Jean-Rémy Falleri, Floréal Morandat, Elyas Ben Hadj Yahia, and Laurent Réveillère. Raters' reliability in clone benchmarks construction. *Empirical Software Engineering*, 22(1):235–258, 2017.

[22] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 175–186, New York, NY, USA, 2014. ACM.

[23] Paul Clements and Linda Northrop. *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[24] Michael L. Collard and Jonathan I. Maletic. srcml, 2017. `http://www.srcml.org`.

[25] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *SoCC '10*, pages 143–154, 2010.

[26] J. R. Cordy. Comprehending reality - practical barriers to industrial adoption of software maintenance automation. In *11th IEEE International Workshop on Program Comprehension, 2003.*, pages 196–205, May 2003.

[27] J.R. Cordy. The txl programming language. `http://www.txl.ca/`.

[28] A. M. Dalgarno. Jump-starting software product lines with clone detection. In *2008 12th International Software Product Line Conference*, pages 351–351, Sept 2008.

[29] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[30] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378–415, March 2000.

[31] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An exploratory study of cloning in industrial software product lines. *17th European Conference on Software Maintenance and Reengineering*, pages 25–34, 2013.

[32] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pages 109–118, 1999.

[33] Slawomir Duszynski, Jens Knodel, and Martin Becker. Analyzing the source code of multiple software variants for reuse potential. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, WCRE '11, pages 303–307, Washington, DC, USA, 2011.

[34] Paul Eggert, Mike Haertel, David Hayes, Richard Stallman, and Len Tower. Diffutils - gnu project - free software foundation. `http://www.gnu.org/software/diffutils`, 2015.

[35] Raimar Falke, Pierre Frenzel, and Rainer Koschke. Empirical evaluation of clone detection using syntax suffix trees. *Empirical Software Engineering*, 13(6):601–643, 2008.

[36] The Apache Software Foundation. Apache hadoop. `http://hadoop.apache.org/`.

[37] The Apache Software Foundation. Apache hive project. `http://hadoop.apache.org/hive`.

[38] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *ICSE '08*, pages 321–330. ACM, 2008.

[39] Mark Gabel, Junfeng Yang, Yuan Yu, Moises Goldszmidt, and Zhendong Su. Scalable and systematic detection of buggy inconsistencies in source code. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 175–190, New York, NY, USA, 2010. ACM.

[40] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. Bigbench: Towards an industry standard benchmark for big data analytics. In *SIGMOD '13*, pages 1197–1208, New York, NY, USA, 2013.

[41] N. Göde and R. Koschke. Incremental clone detection. In *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, pages 219–228, March 2009.

[42] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, Feb 2005.

[43] Sam Grier. A tool that detects plagiarism in pascal programs. *SIGCSE Bull.*, 13(1):15–20, February 1981.

[44] Simon Harris. Simian. `http://www.harukizaemon.com/simian/`.

[45] A. Hemel and R. Koschke. Reverse engineering variability in source code using clone detection: A case study for linux variants of consumer electronic devices. In *2012 19th Working Conference on Reverse Engineering*, pages 357–366, Oct 2012.

[46] Y. Higo and S. Kusumoto. Enhancing quality of code clone detection with program dependency graph. In *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, pages 315–316, Oct 2009.

[47] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: incremental, distributed, scalable. In *2010 IEEE International Conference on Software Maintenance*, pages 1–9, Sept 2010.

[48] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Inter-project functional clone detection toward building libraries - an empirical study on 13,000 projects. In *2012 19th Working Conference on Reverse Engineering*, pages 387–391, Oct 2012.

[49] Tomoya Ishihara, Yoshiki Higo, and Shinji Kusumoto. *How Often Is Necessary Code Missing? — A Controlled Experiment —*, pages 156–163. Springer International Publishing, Cham, 2014.

[50] J. F. Islam, M. Mondal, and C. K. Roy. Bug replication in code clones: An empirical study. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 68–78, March 2016.

[51] Patricia Jablonski and Daqing Hou. Cren: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the ide. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '07, pages 16–20, New York, NY, USA, 2007. ACM.

[52] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE '07*, pages 96–105. IEEE Computer Society, 2007.

[53] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[54] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. Context-based detection of clone-related bugs. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 55–64, New York, NY, USA, 2007. ACM.

[55] J. H. Johnson. Substring matching for clone detection and change tracking. In *Proceedings 1994 International Conference on Software Maintenance*, pages 120–126, Sep 1994.

[56] J. Howard Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, CASCON '93, pages 171–183. IBM Press, 1993.

[57] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. Clonedetective - a workbench for clone detection research. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 603–606, Washington, DC, USA, 2009. IEEE Computer Society.

[58] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on*, 28(7):654–670, Jul 2002.

[59] C. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful. In *2006 13th Working Conference on Reverse Engineering*, pages 19–28, Oct 2006.

[60] I. Keivanloo, C. Forbes, A. Hmood, M. Erfani, C. Neal, G. Peristerakis, and J. Rilling. A linked data platform for mining software repositories. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 32–35, 2012.

[61] I. Keivanloo, C. Forbes, and J. Rilling. Similarity search plug-in: Clone detection meets internet-scale code search. In *2012 4th International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation (SUITE)*, pages 21–22, June 2012.

[62] I. Keivanloo, J. Rilling, and P. Charland. Internet-scale real-time code clone search via multi-level indexing. In *2011 18th Working Conference on Reverse Engineering*, pages 23–27, Oct 2011.

[63] I. Keivanloo, C. K. Roy, and J. Rilling. Java bytecode clone detection via relaxation on code fingerprint and semantic web reasoning. In *2012 6th International Workshop on Software Clones (IWSC)*, pages 36–42, June 2012.

[64] I. Keivanloo, C. K. Roy, J. Rilling, and P. Charland. Shuffling and randomization for scalable source code clone detection. In *2012 6th International Workshop on Software Clones (IWSC)*, pages 82–83, June 2012.

[65] I. Keivanloo, F. Zhang, and Y. Zou. Threshold-free code clone detection for a large-scale heterogeneous java repository. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 201–210, March 2015.

[66] Iman Keivanloo, Juergen Rilling, and Ying Zou. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 664–675, New York, NY, USA, 2014. ACM.

[67] Iman Keivanloo, Chanchal K. Roy, and Juergen Rilling. Sebyte: Scalable clone and similarity search for bytecode. *Science of Computer Programming*, 95, Part 4:426 – 444, 2014. Special Issue on Software Clones (IWSC'12).

[68] Miryung Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in oopl. In *Empirical Software Engineering, 2004. ISESE '04. Proceedings. 2004 International Symposium on*, pages 83–92, Aug 2004.

[69] Miryung Kim and David Notkin. Program element matching for multi-version program analyses. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 58–64, New York, NY, USA, 2006. ACM.

[70] Raghavan Komondoor and Susan Horwitz. *Using Slicing to Identify Duplication in Source Code*, pages 40–56. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[71] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *WCRE*, pages 44–54, 1997.

[72] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1):77–108, 1996.

[73] R. Koschke. Large-scale inter-system clone detection using suffix trees. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 309–318, March 2012.

[74] Rainer Koschke. Large-scale inter-system clone detection using suffix trees and hashing. *Journal of Software: Evolution and Process*, 26(8):747–769, 2014.

[75] Rainer Koschke, Ettore Merlo, and Andrew Walenstein, editors. *Duplication, Redundancy, and Similarity in Software, 23.07. - 26.07.2006*, volume 06301 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[76] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 301–309, 2001.

[77] Daniel E. Krutz and Wei Le. A code clone oracle. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 388–391, New York, NY, USA, 2014. ACM.

[78] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *1997 Proceedings International Conference on Software Maintenance*, pages 314–321, Oct 1997.

[79] Thomas Lancaster and Fintan Culwin. A comparison of source code plagiarism detection engines. 14:101–112, 06 2004.

[80] Thierry Lavoie and Ettore Merlo. Automated type-3 clone oracle using levenshtein metric. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 34–40, New York, NY, USA, 2011. ACM.

[81] Mu-Woong Lee, Jong-Won Roh, Seung-won Hwang, and Sunghun Kim. Instant code clone search. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 167–176, New York, NY, USA, 2010. ACM.

[82] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, March 2006.

[83] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.

[84] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. In *29th International Conference on Software Engineering (ICSE'07)*, pages 106–115, May 2007.

[85] D. Lo and Siau-Cheng Khoo. Quark: Empirical assessment of automaton-based specification miners. In *Reverse Engineering, 2006. WCRE '06. 13th Working Conference on*, pages 51–60, 2006.

[86] Chung-Horng Lung, Marzia Zaman, and Amit Nandi. Applications of clustering techniques to software partitioning, recovery and restructuring. *Journal of Systems and Software*, 73(2):227 – 244, 2004. Applications of statistics in software engineering.

[87] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers. Big data: The next frontier for innovation, competition, and productivity. Technical report, McKinsey Global Institute, 2011, 2011.

[88] J. Mayrand, C. Leblanc, and E. M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *1996 Proceedings of International Conference on Software Maintenance*, pages 244–253, Nov 1996.

[89] D. Mazinanian, N. Tsantalis, R. Stein, and Z. Valenta. Jdeodorant: Clone refactoring. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 613–616, May 2016.

[90] Thilo Mende, Rainer Koschke, and Felix Beckwermert. An evaluation of code similarity identification for the grow-and-prune model. *Journal of Software Maintenance and Evolution*, 21(2):143–169, March 2009.

[91] Cade Metz. The next big os war is in your dashboard, 2015.

[92] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Proceedings Eighth IEEE Symposium on Software Metrics*, pages 87–94, 2002.

[93] Hiroaki Murakami, Yoshiki Higo, and Shinji Kusumoto. A dataset of clone references with gaps. `http://sdl.ist.osaka-u.ac.jp/~h-murakm/2014_clone_references_with_gaps/`.

[94] Hiroaki Murakami, Yoshiki Higo, and Shinji Kusumoto. A dataset of clone references with gaps. In *MSR'14*, pages 412–415, 2014.

[95] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Clone management for evolving software. *IEEE Transactions on Software Engineering*, 38(5):1008–1026, Sept 2012.

[96] Joel Ossher, Hitesh Sajnani, and Cristina Lopes. File cloning in open source java projects: The good, the bad, and the ugly. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 283–292, Washington, DC, USA, 2011. IEEE Computer Society.

[97] Jin-woo Park, Mu-Woong Lee, Jong-Won Roh, Seung-won Hwang, and Sunghun Kim. Surfacing code in the dark: an instant clone search approach. *Knowledge and Information Systems*, pages 1–33, 2013.

[98] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD '09*, pages 165–178, New York, NY, USA, 2009. ACM.

[99] PMD. Cpd. `http://pmd.sourceforge.net/`.

[100] Lutz Prechelt and Guido Malpohl. Finding plagiarisms among a set of programs with jplag. 8, 03 2003.

[101] Tilmann Rabl, Michael Frank, Hatem Mousselly Sergieh, and Harald Kosch. A data generator for cloud-scale benchmarking. In *TPCTC'10*, pages 41–56. Springer-Verlag, 2011.

[102] M. M. Rahman and C. K. Roy. On the use of context in recommending exception handling code examples. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 285–294, Sept 2014.

[103] Damith C. Rajapakse and Stan Jarzabek. An investigation of cloning in web applications. In David Lowe and Martin Gaedke, editors, *Web Engineering*, pages 252–262, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[104] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165 – 1199, 2013.

[105] C. K. Roy and J. R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *2008 16th IEEE International Conference on Program Comprehension*, pages 172–181, June 2008.

[106] C. K. Roy, M. F. Zibran, and R. Koschke. The vision of software clone management: Past, present, and future (keynote paper). In *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 18–33, Feb 2014.

[107] Chanchal K. Roy and James R. Cordy. Towards a mutation-based automatic framework for evaluating code clone detection tools. In *Proceedings of the 2008 C3S2E Conference*, C3S2E '08, pages 137–140, New York, NY, USA, 2008. ACM.

[108] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, May 2009.

[109] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. Technical Report TR 2007-541, School of Computing, Queens University, 2007. 115 pp.

[110] C.K. Roy and J.R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 172–181, June 2008.

[111] C.K. Roy and J.R. Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*, pages 157–166, April 2009.

[112] F. Van Rysselberghe and S. Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pages 336–339, Sept 2004.

[113] R.K. Saha, C.K. Roy, and K.A. Schneider. An automatic framework for extracting and classifying near-miss clone genealogies. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 293–302, 2011.

[114] H. Sajnani, J. Ossher, and C. Lopes. Parallel code clone detection using mapreduce. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 261–262, June 2012.

[115] Hitesh Sajnani, Vaibhav Saini, and Cristina Lopes. A parallel and efficient approach to large scale clone detection. *Journal of Software: Evolution and Process*, 27(6):402–429, 2015. JSME-13-0129.R2.

[116] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1157–1168. ACM, 2016.

[117] N. Schwarz, M. Lungu, and R. Robbes. On how often code is cloned across repositories. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1289–1292, Zurich, Switzerland, 2012.

[118] Chintakindi Srinivas, Vangipuram Radhakrishna, and C.V. Guru Rao. Clustering software components for program restructuring and component reuse using hybrid xnor similarity function. *Procedia Technology*, 12:246 – 254, 2014. The 7th International Conference Interdisciplinarity in Engineering, INTER-ENG 2013, 10-11 October 2013, Petru Maior University of Tirgu Mures, Romania.

[119] M. Stephan. Model clone detector evaluation using mutation analysis. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 633–638, Sept 2014.

[120] J. Svajlenko, I. Keivanloo, and C. K. Roy. Scaling classical clone detection tools for ultra-large datasets: An exploratory study. In *2013 7th International Workshop on Software Clones (IWSC)*, pages 16–22, May 2013.

[121] J. Svajlenko and C. K. Roy. Evaluating clone detection tools with bigclonebench. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 131–140, Sept 2015.

[122] J. Svajlenko and C. K. Roy. Evaluating clone detection tools with BigCloneBench. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 131–140, Sept 2015.

[123] J. Svajlenko and C. K. Roy. Cloneworks: A fast and flexible large-scale near-miss clone detection tool. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 177–179, May 2017.

[124] J. Svajlenko, C. K. Roy, and S. Duszynski. Forksim: Generating software forks for evaluating cross-project similarity analysis tools. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 37–42, Sept 2013.

[125] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 476–480, Washington, DC, USA, 2014. IEEE Computer Society.

[126] Jeffrey Svajlenko, Iman Keivanloo, and Chanchal K. Roy. Big data clone detection using classical detectors: an exploratory study. *Journal of Software: Evolution and Process*, 27(6):430–464, 2015. JSME-13-0126.R1.

[127] Jeffrey Svajlenko and Chanchal Roy. Bigclonebench. `http://www.jeff.svajlenko.com/bigclonebench.html`.

[128] Jeffrey Svajlenko and Chanchal K. Roy. Evaluating modern clone detection tools. In *The 30th International Conference on Software Maintenance and Evolution*, ICSME 2014, page 10, 2014.

[129] Jeffrey Svajlenko and Chanchal K. Roy. Bigcloneeval: A clone detection tool evaluation framework with bigclonebench. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016.

[130] Jeffrey Svajlenko and Chanchal K. Roy. Fast and flexible large-scale clone detection with cloneworks. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 27–30, Piscataway, NJ, USA, 2017. IEEE Press.

[131] Jeffrey Svajlenko, Chanchal K. Roy, and James R. Cordy. A mutation analysis based benchmarking framework for clone detectors. In *Proceedings of the 7th International Workshop on Software Clones*, IWSC '13, pages 8–9, Piscataway, NJ, USA, 2013. IEEE Press.

[132] Teradata. Teradata database - teradata inc. `http://www.teradata.com`.

[133] W. Toomey. Ctcompare: Code clone detection using hashed token sequences. In *Software Clones (IWSC), 2012 6th International Workshop on*, pages 92–93, June 2012.

[134] M. S. Uddin, C. K. Roy, K. A. Schneider, and A. Hindle. On the effectiveness of simhash for detecting near-miss clones in large scale software systems. In *2011 18th Working Conference on Reverse Engineering*, pages 13–22, Oct 2011.

[135] Md. Sharif Uddin. Dealing with clones in software: A practical approach from detection towards management. Master's thesis, University of Saskatchewan, Saskatoon, Saskatchewan, February 2014.

[136] M.S. Uddin, C.K. Roy, and K.A. Schneider. Simcad: An extensible and faster clone detection tool for large scale software systems. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 236–238, May 2013.

[137] A. Walenstein, N. Jyoti, Junwei Li, Yun Yang, and A. Lakhotia. Problems creating task-relevant clone detection reference data. In *WCRE*, pages 285–294, 2003.

[138] Andrew Walenstein and Arun Lakhotia. The software similarity problem in malware analysis. In Rainer Koschke, Ettore Merlo, and Andrew Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[139] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. Searching for better configurations: A rigorous approach to clone evaluation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 455–465, New York, NY, USA, 2013. ACM.

[140] WBDB. 5th workshop on big data benchmarking. `http://clds.sdsc.edu/wbdb2014.de`.

[141] Lin Ye and Guoxiang Yao. Parallel clone code detector in mapreduce. *Journal of Software*, 9(6), 2014.

[142] Minhaz F. Zibran and Chanchal K. Roy. A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring. In *Proceedings of the 2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, SCAM '11, pages 105–114, Washington, DC, USA, 2011. IEEE Computer Society.

[143] Minhaz F. Zibran and Chanchal K. Roy. The road to software clone management: A survey. Technical Report TR 2012-03, Department of Computer Science, University of Saskatchewan, 2012. 62 pp.