Large-scale analysis of the security of cryptographic keys

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

By
Ronald Rivera

Permission to Use

In presenting this thesis in partial fulfillment 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

Or

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9
Canada

Abstract

Cryptographic algorithms are considered provably secure due to their strong mathematical foundation. Notwithstanding, real-life application of cryptographic algorithms and protocols continues to fail. These failures are frequently due to low entropy, faulty library implementation, and Application Programming Interface (API) misuse. Biases introduced during the generation process incorporate distinct bit patterns in RSA cryptographic keys allowing their attribution, thus endangering their advertised security.

This thesis proposes a novel attribution approach to link cryptographic keys to their originating libraries based on moduli's characteristics. We analyze over 6.5 million generated keys and show that only a few of these characteristics are enough to achieve a 75% accuracy in the attribution of individual keys to their originating library. Also, depending on the library, our approach is sensitive enough to pinpoint the corresponding major, minor, and build release information for several libraries with accuracy levels between 81% and 98%. We further explore the attribution of SSH keys collected from publicly facing IPv4 addresses proving that our approach differentiates individual libraries of RSA keys with a 95% accuracy.

Acknowledgements

I wish to express my gratitude to my adviser, Natalia Stakhanova, for her guidance and uncountable suggestions to improve my work quality. Thank you for your never-ending support and for always being there when we needed you. Because of your patience and guidance, I learned so much more than I could ever have imagined.

I also wish to thank our laboratory technician, Enrico Branca, for all the technical knowledge and cyber-security perspectives he shared with me while working on different projects. His support contributed a great deal to the simplification and understanding of complex concepts.

I thank my friends for all those unforgettable moments watching movies, going out, eating pizza or chicken wings or merely texting or talking on the phone. Whether in Canada, the US or El Salvador, you all cheered me up and encouraged me to do my best all the time.

I thank my family for their unconditional love and support throughout these years. They were always there when I needed any advice or just talking to someone when things did not result as expected. They always believed in me and gave me the strength to keep on doing the best I could, regardless of those difficulties that life throws at you. Even though we were apart, you made me feel as if I had never left. I love you all.

To all of you, thank you. I would not have made it without you.

To my mom, the loveliest rose of them all.

Contents

List of Tables

List of Figures

List of Abbreviations

PKCS	Public-Key Cryptography Standards
SSH	Secure Shell Protocol
SSL	Secure Sockets Layer Protocol
TLS	Transport Layer Security
GnuTLS	GNU Transport Layer Security
GPG	GNU Privacy Guard
PGP	Pretty Good Privacy
DTLS	Datagram Transport Layer Security
RSA (cryptosystem)	Rivest–Shamir–Adleman (cryptosystem)
HTTPS	Hypertext Transfer Protocol Secure
API	Application Programming Interface
IP (address)	Internet Protocol (address)
NIST	National Institute of Standards and Technology
LZMA	Lempel–Ziv–Markov chain algorithm
TruRNG	Hardware Random Number Generator
NeuG	True Random Number Generator
FIPS	Federal Information Processing Standards
PEM	Privacy-Enhanced Mail

1 Introduction

Nowadays, above 50% of Internet traffic is encrypted, and the adoption of cryptographic protocols has been increasing over the past years [3]; therefore, secure communications over the Internet has become the norm. Theoretically, cryptography is provably secure due to its strong dependence on mathematics. In practice, however, communications' security depends on how developers and applications implement encryption standards.

Over the past decade, several studies identified alarming weaknesses affecting the security of widely used communication protocols that depend upon cryptographic implementations such as TLS/SSL [29], SSH [26] and HTTPS [21, 28, 4]; most of these studies reasoned that inadequate security levels result from weakly generated keys. Furthermore, some of these studies traced the problem back to weak random number generators and lack of entropy [29, 24, 16]. Additionally, others observed the improper implementation of cryptographic libraries [22, 41, 56, 45] and the pure misuse of cryptographic algorithms, for example, RSA private keys mistakenly left embedded in binary files [23].

These security issues lead to the development of techniques that allow the identification of insecure and vulnerable keys. From these, the vast majority focused on the analysis of binaries that contain vulnerable keys [42, 41, 52, 49], cryptographic libraries and the APIs that produce them [47, 56].

The improper implementation of the RSA algorithm and its related cryptographic operations cause software libraries to produce keys with distinguishable patterns that can identify the affected cryptographic keys' source. In this regard, Svenda et al. [56] were the first to observe that implementation decisions in the presence of specific hardware are sufficient to associate RSA keys with their corresponding software library; Nemec et al. [47] and Janovsky et al. [32] improved upon this work.

The attribution performed by both approaches relied on rules based on the libraries' biases. Despite the high accuracy achieved by Nemec et al. [47], both methods failed to attribute individual keys to their corresponding library. This results from their approach only considering the most significant byte and the frequency of all their possible combinations; thus, both methods can only attribute groups keys to closely related libraries. Therefore, classifying individual keys remains unexplored and has real-life applications that help identify weak cryptographic keys and, consequently, vulnerable implementations.

This research introduces a new approach that allows for the attribution of individual RSA keys based on features extracted from a key's modulus. These features resulted from analyzing the RSA public keys' characteristics and determining how much information is needed to correlate individual keys with the library that produced them. This new solution takes advantage of the fact that, in practice, cryptographic keys seem

to have distinguishable bit patterns that make possible attributing individual keys to their corresponding originating library.

Our validation experiments on over 6.5 million generated keys show that we can accurately attribute an individual key to the originating library with 75% accuracy with only a few modulus characteristics. We are further able to accurately attribute keys to the corresponding major, minor, build and in some cases patch releases[1] for several libraries achieving accuracy in the range of 81% - 98%.

Our findings suggest that code changes applied to some library implementation within a major, minor, or even build versions leave significant traces in the generated keys, unfortunately, allowing for source attribution.

We compare our approach to the most recent study by Nemec et. al [47]. Their previous work was able to accurately (94% accuracy) attribute the keys to the groups of similar libraries. We show that our approach outperforms their technique providing a more granular attribution to an individual library and its version.

We further explore the source attribution of almost 200,000 RSA keys collected from publicly facing IPv4 addresses. Our analysis of these collected keys shows that they generally come from a homogeneous pool of libraries (e.g., Dropbear and OpenSSH). Moreover, we can differentiate individual libraries of RSA collected keys with 95% accuracy. For particular library versions, we obtained 68% to 100% accuracy for most libraries having enough keys. More importantly, we have been able to do this without any prior knowledge of the system or the library that generated them.

1.1 Contributions

Our research offers the following concrete scientific contributions:

- **Identification of moduli's bit-patterns that allow for high accuracy during the attribution process:** Our work strongly leverages several bit-patterns found in RSA moduli to associate software implementations with individual keys. Moreover, we demonstrate that highly accurate attribution is possible using only a few moduli characteristics.

- **Attribution of individual RSA keys through a small set of features:** The approach we present in this research allows the association of individual RSA keys with their source libraries and the corresponding library version, whether implementing hardware random number generators or not. We provide evidence that the solution works across multiple cryptographic libraries such as OpenSSH, OpenSSL, GNU TLS and GPG and demonstrates that attribution of real-life keys is possible by accurately associating real-life keys collected in the wild from Canadian IPv4 hosts with their respective libraries and versions.

[1] We refer to library versions using the conventional notation of software versioning represented by major.minor[.build[.patch]] (e.g., 1.0.2k)

- **Implementation of a prototype for the proposed attribution approach:** In addition to the theoretical framework supporting our approach, we provide the source code for obtaining and generating cryptographic keys, and extracting their textual and numerical features. Besides, we also provide features set used in the attribution process.

- **Profile the presence and usage of vulnerable RSA cryptographic keys used by the SSH protocol across Canada:** To validate our proposed technique's correctness and soundness, and its capacity to generalize to previously unknown cryptographic keys, we scanned the Canadian Internet space and collected almost 200,000 SSH keys; we then applied our solution to this set of keys and attributed their originating libraries and, when possible, the corresponding library version.

1.2 Thesis Structure

We organized our research's content in seven chapters, as follows:

- **Chapter 1: Introduction.** This chapter introduces our research's general overview while highlighting its importance and specific contributions to the scientific field. Also, we briefly aboard two papers closely related to ours and their shortcomings. Finally, we provide the results achieved by our proposed approach.

- **Chapter 2: Background.** This chapter provides a quick introduction to the Public Key Cryptography and the RSA cryptographic algorithm. We briefly describe the SSH, SSL and TLS protocols, including the software libraries that depend on them.

- **Chapter 3: Related Work.** This chapter presents a concise literature review on several studies about the root causes causing real-world application of cryptographic protocols and libraries to fail. Additionally, we examined research on cryptographic keys' attribution.

- **Chapter 4: Methodology.** This chapter gives a step-by-step explanation of our data collection process and how we extracted useful features from the collected cryptographic keys and their meanings. Furthermore, we describe the tests we used to analyze the data and define the Machine Learning models implemented in the attribution process.

- **Chapter 5: Validation Study.** This chapter presents the experimental setup used to obtain the ground truth to validate our new approach's accuracy and soundness. Here, we show how we generated cryptographic keys, clearly explain the DieHarder tests and results, and show the parameters used in the Machine Learning Models to attribute RSA keys correctly. Moreover, we compare our results with the results obtained by Nemec et al. [47].

- **Chapter 6: Wide Internet Scan of the Canadian IPv4 Space.** In this chapter, we offer a generalization of our approach. We collected 191,975 SSH keys retrieved from publicly available Canadian websites and show how we used our approach to attribute their SSH keys.

- **Chapter 7: Conclusions.** This chapter presents the conclusions drawn after parsing, analyzing and attributing cryptographic keys originating libraries and operating systems. Additionally, we suggest new and exciting research opportunities where our approach can be employed to derive or extend our research.

2 Background

This chapter provides the theoretical background that servers as a base for our research. We summarize the inner working of the public-key cryptography, the RSA cryptosystem and related protocol and libraries.

2.1 Public-Key Cryptography

Diffie and Hellman proposed the public-key cryptography on an argumentative paper [19] introducing the idea of using two cryptographic keys, one for encryption and the other for decryption. Later, they concretized the mathematical and computational requirements for their approach [20]. Their model eliminated the keys distribution problem while providing users with the ability to encrypt and decrypt messages in a computationally inexpensive manner.

Public-key cryptography's security depends on the difficulty of solving equations involving logarithms over a finite field of prime elements; this strong dependency on mathematics makes cryptanalysis computationally infeasible for attackers with limited resources. Consequently, public-key cryptography is not unconditionally secure but rather computationally secure since attackers with sufficient computational power can factorize cryptographic keys.

Generally speaking, all cryptographic systems transform plaintext into its corresponding ciphertext. On this matter, the public-key cryptosystem proposed by Diffie and Hellman provides a way to encrypt and decrypt data with a set of two different cryptographic keys known as the private and the public Keys. The private Key must remain undisclosed at all times, and the public Key made publicly accesible by all users.

Having public keys is beneficial to systems with numerous users since it reduces the number of keys needed to start private and secure communications. For instance, a system with x number of active users requires $\frac{x^2-x}{2}$ keys when working with asymmetric cryptography. However, in public-key cryptography, the number of cryptographic keys is reduced to x, one public key per user.

Lastly, the public-key cryptosystem proposed by Diffie and Hellman provides users with privacy, authentication, and non-repudiation properties. Privacy is achieved when users encrypt messages with the receiver's public key; this ensures that only the intended recipient can decrypt them with the associated private key. In the form of digital signatures, authentication occurs when the receiver party applies the sender's public key, along with a verification algorithm, on a message signed by the sender. Finally, suppose private keys have not been stolen or revoked. In that case, malicious parties cannot repudiate neither encrypted nor signed messages since legitimate users are the only ones in possession of their private keys.

2.2 The RSA Cryptosystem

Rivest, Shamir, and Adleman introduced the first working implementation of public-key cryptography [54]. Nowadays, the RSA algorithm is widely used worldwide by applications and devices to keep communications private and secure by implementing encryption, digital signatures, and certificates. Users implementing RSA need to calculate the product of two random primes: $n = p * q$, where n is the *RSA modulus*. These primes must be secret and sufficiently large so that factorizing the modulus by performing an exhaustive search is infeasible even for adversaries with high computational power.

Then, users compute $\phi(n) = (p-1) * (q-1)$ and selects a public exponent e that must be large, random and coprime with $\phi(n)$ such that the greatest common divisor between them is equal to one. Finally, the user computes a private exponent $d = e^{-1} \bmod \phi(n)$. In RSA, the pair of positive integers (e, n) is the **public key** Pub_k, where e is the RSA *public exponent*. The pair of positive integers (d, n) is the **private key** $Priv_k$. After calculating both the private and public keys, the encryption and decryption processes are trivial. For encrypting a plaintext m, users calculate $c = m^d \bmod n$ and send c to the receiver entity who decrypts the message by computing $m = c^e \bmod n$.

2.3 Protocols and Libraries implementing RSA

Many cryptographic libraries and communication protocols are using the RSA cryptosystem. For this research, we explored four widely used open-source libraries: OpenSSH, OpenSSL, GnuTLS and GPG. In this section, we describe each library and the SSH, SSL and TLS underlying communication protocols.

2.3.1 The Secure Shell Protocol (SSH)

The Secure Shell Protocol (SSH) uses public-key cryptography to provide a secure channel so that users communicate securely with remote machines through insecure networks [60]. Clients typically connect through port 22; occasionally, the SSH servers use other ports such as 2222. Once an encrypted channel is open, users can request services, run commands, or log in to remote systems.

The SSH protocol has two major versions referred to as *SSH version 1.x* and *SSH version 2.x*. Both versions can run in Unix-like (i.e., Linux and macOS) and Windows environments. Some SSH servers provide backward compatibility to clients running the protocol's version 1.x. When a server understands both versions, it advertises its *protoversion* as SSH version 1.99.

The SSH protocol has three principal components. First, The Transport Layer Protocol [61] provides users with authentication, confidentiality, integrity, and data compression capabilities; it commonly runs over a TCP/IP connection. Second, The User Authentication Protocol executes over the Transport Layer Protocol and is a general-purpose protocol that allows users to authenticate themselves in a remote server securely. Third, Connection Protocol. It manages the connections by multiplexing several logic channels

from the encrypted tunnel and runs over the User Authentication Protocol.

2.3.2 The Secure Sockets Layer Protocol (SSL)

The Secure Sockets Layer protocol [6] is an application protocol independent and provides privacy and reliability between communicating parties through an initial handshake to define a secret key; thus, securing a connection channel between them. SSL runs on top of the Transmission Control Protocol [50] or other reliable transport protocol; it uses symmetric cryptography for privacy and public-key cryptography for authentication. The SSL protocol also provides reliability to its users by implementing a keyed Message Authentication [37] code and secure hash functions to verify that the information transferred between has not been altered or corrupted by adversaries or any other means.

2.3.3 The Transport Layer Security Protocol (TLS)

The Transport Layer Security protocol [53] is a replacement for the Secure Sockets Layer protocol and provides a secure communication channel over insecure networks for interacting parties. It requires that the underlying transport protocol be reliable and in-order data stream. By default, TLS provides server-side authentication via symmetric cryptography or public-key cryptography. Data sent by using TLS is encrypted point-to-point, which provides confidentiality and integrity of the transmitted data.

2.3.4 The OpenSSH Library

The OpenSSH library [1] is a network utility based on the SSH protocol and provides a secure communication channel in client-server architectures; users commonly use it to log in to remote servers. It is integrated into Unix-like and Windows environments and helps prevent eavesdropping and session hijacking by encrypting traffic from point-to-point by replacing insecure protocols like Telnet and FTP with more trustworthy alternatives (e.g., the SFTP and ssh commands).

2.3.5 The OpenSSL Library

The OpenSSL [2] is an open-source general pupose cryptographic software library that runs over the Transport Layer Security and the Secure Sockets Layer protocols; therefore, it secures communication channels on insecure networks. For instance, it widely implemented by websites offering HTTPS connections to authenticate themeselves to users while providng privacy and data integrity. It is available on Unix-like and Windows environments.

2.3.6 The GnuTLS Transport Layer Security Library

The GnuTLS [57] is an open-source communication library that implements the SSL, TLS and DTLS protocols in a work-out-of-the-box manner; this characteristic eliminates the complexity of implementing cryptographic

algorithms. It is available for Unix-like and Windows operating systems under x86, x86-64 and ARM architectures. It runs over the Network Transport Layer and makes easy handling cryptographic protocols and structures, such as X.509, PKCS 12 and OpenPGP.

2.3.7 The GNU Privacy Guard (GPG) Library

The GNU Privacy Guard [35] is an open-source command-line tool that completely fulfills the OpenPGP standard [14] and enables users to encrypt and digitally sign data and communications. It is available for Unix-like and Windows environments and completely integrates with Microsoft Outlook, KMail and other OpenGPG-compliant systems. Furthermore, it correctly implements symmetric-key cryptography and public-key cryptography for secure key exchange.

3 Related Work

This chapter provides a succinct summary of previous research concerning RSA keys' attribution and human errors affecting cryptographic systems, software development, network devices, and operating systems.

3.1 Human errors affecting software and devices

Anderson examined cryptographic systems failures in the United Kingdom's financial sector [5]. He showed that fraud mostly happened due to implementation errors and management failures. Among these problems, he identified weaknesses in PIN creation, transmission and storing of unencrypted confidential data, programming errors, backdoors and weak RSA keys. Almost 30 years later, Køien reasoned that most security problems we face nowadays are similar to those expressed by Anderson [36]; cryptanalysis is therefore not the leading cause of cryptographic systems' failures.

Lazar et al. analyzed 269 cryptography-related vulnerabilities [39]; they estimated that 83% had their root cause in implementation and programming errors. Their work showed that insufficient entropy when generating RSA keys, the transmission of unencrypted data and poorly implemented cryptographic solutions could result in stolen logging credentials, factorization of cryptographic keys or impersonation of trusted sources.

Wang et al. investigated the implications of how compilers optimize undefined behaviour in C code [58]. They noticed that some developers are unaware of optimizations performed; thus, shipping their software with signed integer overflows, broken boundary checks, useless software patches, reduction of entropy levels, and sometimes even allowing privilege escalation. From their part, Yin et al. studied 720 software patches intended to solve memory leaks, buffer overflows, and race conditions on Operating Systems [59]. They discovered that around 10% of patches did not work, and instead, they introduced new bugs resulting in software crashing, data corruption and destruction, or new security issues. Furthermore, they marked the issuing of multiple patches failing to solve the same bug.

Costin et al. conducted a static analysis of 32,356 firmware discovering 38 unknown vulnerabilities affecting nearly 123 devices [16]. These findings included hardcoded credentials and backdoors. They also identified devices running on vulnerable software and firmware images and web servers running with root privileges. Specifically to RSA, they discovered several devices containing private keys; this could enable the impersonation of entities and communications decryption.

3.2 Entropy Issues and Factorization

Everspaugh et al. analyzed Linux's random, urandom and get_random_int interfaces [24], commonly used to obtain random values used for security and cryptography. They proved the existence of boot-time entropy holes in virtualized environments producing deterministic or repeated results during booting-time or when resuming from a snapshot. For instance, outputs from random and urandom could be deterministic due to virtual machines initializing memory pages with zeroes. Moreover, get_random_int will still produce results obtained from insufficient entropy. These directly impact RSA keys; for example, when virtual machines resume from a snapshot, OpenSSL will generate the same prime p, which is enough to factorize RSA moduli regardless of their size.

Heninger et al. analyzed 5.8 million TLS certificates and 6.2 million SSH keys, finding problems related to the boot-time entropy holes, mostly affecting network devices in the wild [29]. They discovered TLS hosts using default and shared certificates or keys; 5.23% and 5.57%, respectively. Furthermore, 9.60% of SSH hosts were also sharing keys. Keys generated with insufficient entropy allowed the authors to factorize 0.50% of TLS certificates and 0.03% of SSH keys.

Hasting et al. analyzed 37 vendors' behaviour after being notified about vulnerabilities in their devices [28]. Only one vendor released a software update promptly, five released security advisories, ten did not respond at all, and the rest only privately replied. Although some vendors fixed these vulnerabilities in newer devices, vulnerable devices continued to be commercialized, some transitioned between vulnerable and non-vulnerable states and the ones already deployed were not recalled; in some cases, newer devices shipped with new vulnerabilities. On the other hand, due to insufficient entropy affecting moduli's resilience, Hasting et al. factorized 0.37% of 81.2 million RSA unique moduli due to the implementations calculating moduli from a shared prime p or a small list of primes. They also discovered that bit errors affect the moduli's robustness making p and q's length different. Finally, they attributed keys as likely being OpenSSL-generated when p - 1 was not divisible by the first 2048 primes; they claim that 95% of broken certificates had OpenSSL-generated keys.

Li et al. worked with 25 x86/64 executables to develop K-Hunt, a tool for identifying insecure cryptographic keys [41]. Their tool identifies insecure keys by dynamically analyzing cryptographic blocks to determine which data is accessed, how keys are generated, propagated and used. For instance, K-Hunt can determine if the executables generate keys from deterministic inputs or when memory is not immediately clean after cryptographic operations. Their results showed that only three of 25 binaries did not display weak keys.

Durumeric et al. analyzed Certificate Authorities and identified that 683 non-commercial entities issued 80% of 1832 certificates [21]. Moreover, they discovered that only ten countries are responsible for 95% of trusted certificates and 98% of host certificates. Furthermore, Symantec, GoDaddy and Comodo controlled 75% of trusted certificates. Over 99% of trusted leaf certificates used RSA keys, but 50% of them had insecure

keys within their trust chain. For instance, 13% of them signed with weaker keys, having smaller sizes than those they contained within. More than 70% of certificates used 1024-bits RSA keys, and 57% of roots using this size had signed certificates; NIST recommended discontinuing the use of 1024-bits RSA keys in 2016. The authors also found certificates using 512 and 768-bits keys, which were also considered insecure at the time.

Bhargavan et al. studied the Keyless SSL protocol's security implemented by the Content Delivery Network architecture [10]. Cloudflare uses this architecture to improve the performance by redirecting clients to servers near their geographical location. However, the lack of forward-security in TLS-RSA connections makes it possible for an adversary that compromises at least one private key to decrypt recorded connections from arbitrary servers located in different geographical locations. Furthermore, origin servers would sign values from an edge server even if they did not have a valid key exchange message. These attacks could be extended due to session resumptions from compromised edge servers until a compromised session expires. To fix these security issues, the authors proposed that edge servers act only as a medium to relay handshake information between the clients and the origin server.

3.3 Key Source Attribution

Svenda et al. tested cryptographic libraries and cards' randomness and resistance to test if they comply with quality and security expectations against well-known RSA attacks [56]. They assessed the viability to attribute RSA keys' origin from moduli's characteristics. They discovered that it is possible to obtain the intervals used to generate primes based selection restrictions such as the minimal size between $p - q$, $p > q$ or $q > p$, rejection of small factors for $p - 1$ or the primes' distribution differing from randomly generated numbers. Their results showed an accuracy of over 73%, concluding that attribution is possible due to software and smartcards' differences in design, implementation choices and the use of faulty random number generators. Moduli distribution reveals the implementations' underlying algorithm for prime selection. Nevertheless, it is not possible to identify two sources sharing the same or similar algorithms.

Nemec et al. attributed RSA keys' sources based on cryptographic libraries' biases when selection primes [47]. They identified that efficiency improvements, implementation choices and bugs are the leading causes of biases when selecting primes; thus, these biases can sometimes be observable from the moduli. For instance, when identifying the primes' numerical ranges or when cryptographic implementations avoid specific values. In a recent paper [33], Janovsky et al. extended the technique in [56] to attribute cryptographic keys when the private key is known. Even though the work in [47, 33] improved the attribution's accuracy compared to [56], neither of the approaches could attribute keys to individual libraries within groups.

4 Methodology

In this chapter, we explain the steps we undertook during our research. We describe the tests we conducted, and their importance, the features we extracted from the generated and collected RSA moduli and discuss the Machine Learning models we implemented.

4.1 General overview of the approach

From a high-level perspective, our research follows four steps with clearly defined tasks that facilitate the collection, parsing and analysis of RSA keys' characteristics. These steps cover the extraction of features, security tests performed to cryptographic keys and the attribution process. In Figure 4.1, we show how these steps relate to each other.

The general overview of these four steps is as follows:

- **Parsing.** In this step, we derive information from the primary components used in the analysis: metadata, exponents and moduli. We look for repetitive patterns within the moduli's binary representation and obtain the exponents and moduli's security characteristics. Then, we generate metadata related to the keys' generation/collection processes (e.g., data and software implementation).

- **Analysis.** In this step, we examine the exponents and moduli's security expectations. Concretely, we estimate the moduli's randomness, collect and analyze bit-patterns resulting from poor library implementation practices.

- **Feature extraction.** In this step, we transform the parsed data from the generated and collected datasets into feature vectors. We format these features to fit the machine learning algorithms' specifications (e.g., strings converted into categories).

- **Classification.** In this step, we use features extracted in the feature extraction step and feed them into six machine learning classifiers to attribute individual keys to their specific libraries.

4.2 Parsing

We used the ssh-keygen command-line tool to generated SSH public keys in PUB format. Generally speaking, PUB files have the following format:

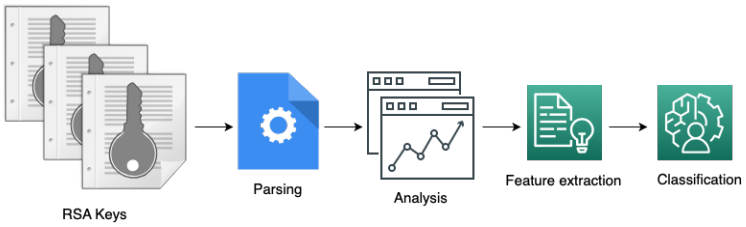


Figure 4.1: The flow of the analysis

format ssh-public-key-base64 *[comments]*

Where *format* is any of the identifiers shown in Table 4.1; it defines the key's encoding and how to interpret the certificates. All these formats are optional, except for ssh-dss and ssh-rsa, the former required if no other encoding is selected and the later recommended. Nevertheless, it is possible to add additional formats by following the specifications detailed in [60] and [40]. The *[comments]* section represents optional comments that parsers commonly ignore when eliciting the public key. PUB files do not store informative data about the key's type (e.g., RSA) since this is obtained from a known source or through an algorithm negotiation.

To obtain the RSA keys' modulus and exponents from PUB files, we automated the decoding process shown in Figure 4.2.

Table 4.1: Public-Key and/or certificates formats for PUB files

Identifier	Status	Type	Content
ssh-dss	Required	Sign	Raw DSS Key
ssh-rsa	Recommended	Sign	Raw RSA Key
pgp-sign-rsa	Optional	Sign	OpenPGP certificates with RSA Key
pgp-sign-dss	Optional	Sign	OpenPGP certificates with DSS Key

Cryptographic keys, including RSA keys, are stored and exchanged in PEM format [43]. The Privacy Enhancement for Electronic Mail (PEM) provides end-to-end enhanced privacy and interoperable security mechanisms between systems. Even though the PEM format did not become widely accepted and was later deemed obsolete due to the PGP and S/MIME general adoption, its encoding became quite popular. For this reason, the Network Working Group formalized it as a de-facto standard for cryptographic keys in [34].

Nevertheless, the SSH protocol does not fully meet the PEM format. Instead, SSH implementations must follow the specifications in [25] since it groups subtle differences that one must consider when working with SSH public keys. For example, the encapsulation boundary strings and each line's width differs in both specifications. There are also other small differences, but those are not relevant for the decoding phase.

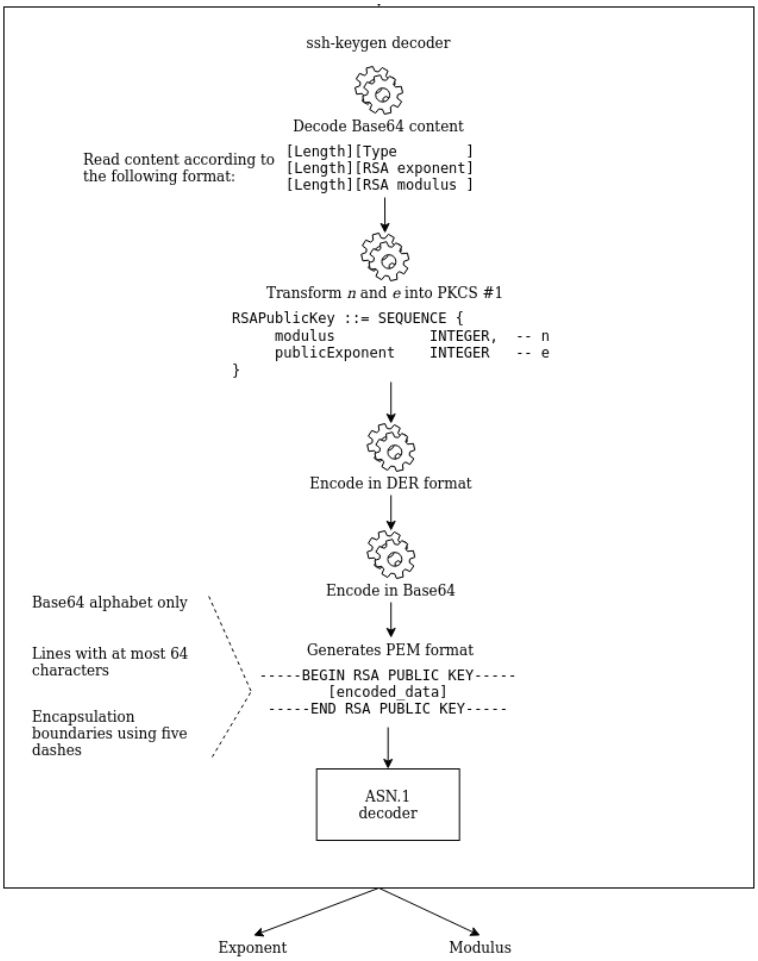


Figure 4.2: The decoding process.

4.3 Analysis

To understand the potential nature of repetitiveness, we performed a statistical analysis of public keys. We adopted the Dieharder testing suite to check the randomness of the available keys.

The Dieharder testing suite [13] is an open-source project that helps to evaluate random number generators. The suite combines improved and independently rewritten versions of the NIST's Statistical Test Suits [8], and additional tests created by the Dieharder's author and other developers. The Dieharder suite includes 31 tests; we give an overview of each test in Table A.1. These tests serve as an indication of whether the keys satisfy the cryptographic security expectations.

Besides the Dieharder tests, we considered the NIST recommendations to check if all gathered and collected RSA moduli and exponents comply with their security expectations. Failing to comply with the following essential characteristics would result in the generation of insecure factorable RSA keys:

- **The RSA modulus' size in bits is equal to or greater than 2048 bits.** The current NIST recommendations specify that suitable moduli are equal to or greater than 2048 bits. Not all non-

acceptable moduli are yet factorable, but the recommendation aims to provide users with a reliable security level to withstand brute-force attacks from adversaries with enough computational power.

- **The RSA modulus is a prime number.** The NIST recommendations state that all moduli must be a prime number to avoid factorization; thus, we check compliance with this recommendation by running the Miller-Rabin Primality Test, which is an efficient algorithm that uses repeating squaring and runs in polynomial time with a complexity of $O(k\ log^3\ n)$.

- **The RSA exponent is a positive integer that is larger than one.** Mathematically, RSA implementations can provide ciphertexts regardless of the exponent's value. Nevertheless, from a security perspective, those cases when the exponent is equal to one mean that no encryption exists; in other words, the ciphertext is always equal to the cleartext.

- **The RSA exponent is a prime number.** The NIST recommendations also stipulate that the chosen exponent must be coprime with $\phi(n)$ to avoid sharing a factor. We check this condition by running the Miller-Rabin primality test the same way as we did for the modulus. Excepting for the size of the number we are checking, there is no difference in how we ran the test.

- **The RSA exponent is an odd number, equal to or larger than 65,537 but lower than 2^{256}.** RSA implementations using an exponent lower than 65,537, particularly the value 3, could be prone to attacks in the absence of proper padding. In these attacks, an adversary could compute the private key based on the message's size or when the number of messages sent is at least equal to the exponent's value. Exponent values larger than 2^{256} add unnecessary overhead when encrypting or verifying digital signatures.

4.4 Feature extraction

For each key, we consider its numerical value and the string representation of its binary equivalent. Both representations are useful to describe the keys' security characteristics to determine if software libraries introduce weaknesses during the key's generation process.

Numerical representation. Each key is decoded into its numerical representation (i.e., exponent and modulus) and subjected to a series of tests to quantify its randomness. The tests' results help describe the key's spatial characteristics to get its position within the numerical spectrum and the likelihood that a particular library may have generated the key.

We derived the following numerical features:

- *Modulus' characteristics*: size in bits and primality.

- *The Shannon Entropy of a modulus*: the presence of entropy is one of the necessary tests of the random number generation process.

- *The cutoff value characteristics*: most algorithms for generating RSA modulus set bits in certain positions; usually, libraries set the first two bits and the last bit to 1[1] which defines/reduces the numerical space for selecting a potential modulus. The *cutoff value* represents the minimum value that a library can ever generate for a modulus in a numerical space. We calculate the cutoff value and check its position against the generated modulus' actual integer value; we refer to it as its *offset* position.

- *Bin position*: Given the fact that the numerical space available for each key is defined by the size in bits, to compare keys of different sizes, we divide the numerical space into 100 sections and assign a positional value to each key depending on where it falls in respect to its relative section.

  An analysis of the spatial characteristics of a key's modulus (i.e., the bin value together with the cutoff and its position) allow us to identify not only the size of the theoretical numerical space used by the library but also compare the relative position of each key with the position of all other keys that share same characteristics and belong to the same numerical space.

- As an estimator of string randomness, we employ both the *Brotli* [31] and the *Lempel–Ziv–Markov chain (LZMA) compression* [51] algorithms. We express the *degree of randomness* as the *compression ratio of a compressed file versus compression ration of an uncompressed file.* The compression ratio provides a quick way to visualize the library's randomness when generating a key.

Textual representation. To further characterize each key, we convert a key's content from its original binary format into a textual format. Then we subject this new representation to a series of tests to identify potential patterns and textual artifacts.

We derive the following types of textual features:

- *Longest repeated substring (LRS) characteristics*: we estimate the presence and the corresponding characteristics of the longest repeated substring patterns within each modulus.

- *0s & 1s characteristics*: we determine and profile the maximum length of continuous zeroes or ones within the string representation of a modulus.

- *Most significant bit (MSB) and Least significant bit (LSB) characteristics*: starting from the first eight bits, we perform a comparative analysis of a modulus' MS versus its LSB.

- *Characteristics of mirror patterns*: starting from the first eight bits, we determine and profile the presence of the same bit-patterns in both the second half of a modulus and the rest of the modulus' binary string minus the pattern's length.

The complete set of features extracted for each key is given in Table 4.2.

[1]For example, setting last bit to 1 ensures that the number is odd

Table 4.2: The features extracted for attribution of generated and collected keys

Feature	Feature Type	Description
Port	Protocol-related	Port used to perform the connection
Protocol	Protocol-related	SSH protocol used; obtained from the SSH Key
Banner	Protocol-related	Banner information for the SSH key
HPN *	Protocol-related	High Performance SSH/SCP – HPN-SSH; SSH patch to improve performance in message exchange
SSH Key	Protocol-related	Actual SSH key in Base64 format
Modulus size	Numerical	Integer value of the modulus size
Is modulus prime?	Numerical	Check if the integer value representing a modulus is prime
Shannon entropy *	Numerical	The Shannon Entropy of a RSA modulus
Brotli compression*	Numerical	Percentage of compression using the Brotli algorithm for text compression
LZMA compression*	Numerical	Percentage of compression using the LZMA algorithm for text compression
Offset	Numerical	Cutoff value for moduli of keys that are generated with standard logic
Offset position	Numerical	Check if the integer value of the modulus is lower than the integer value of the offset
Bin *	Numerical	Which bin, between the range of 1 and 100, the value of n falls in (i.e., first occurrence when n < bin)
Longest repeated substring (LRS) pattern *	String	Actual longest repeated substring pattern within each string representation of a modulus
LRS length	String	Length of LRS pattern
LRS position *	String	Index positions in which LRS pattern is present within each string representation of a modulus
LRS Percentage	String	The percentage that the combined LRS pattern * LRS Pos represents vs the total length of the string representation of a modulus
Zeroes length	String	Maximum length of continuous zeroes within the string representation of a modulus
Zeroes position *	String	Index positions in which the continuous zeroes are present within each string representation of a modulus (list)
Zeroes percentage	String	The percentage that the combined Zeroes Len * Zeroes Pos represents vs the total length of the string representation of a modulus
Ones length	String	Maximum length of continuous ones within the string representation of a modulus
Ones position *	String	Index positions in which the continuous ones are present within each string representation of a modulus (list)
Ones percentage	String	It is the percentage that the combined Zeroes Len * Zeroes Pos represents vs the total length of the string representation of a modulus
Most significant bit (MSB) - Least significant bit (LSB) pattern *	String	Starting from the first eight bits, checking if the first x bits are equal to the last x bits, in the same order (left to right)
MSB LSB length	String	Length of the largest *Most significant bit* vs *Least significant bit* string
MSB LSB percentage	String	It is the percentage that MSB LSB string represents vs the total length of the string representation of a modulus
Mirror pattern *	String	Starting from the first eight bits, checking if the first x bits are present on the second half of a modulus (regardless of their position);
Mirror length	String	Length of the largest Mirror pattern
Mirror position*	String	Index positions in which Mirror pattern is present within each string representation of a modulus;
Mirror percentage	String	It is the percentage that Mirror pos * Mirror len represents vs the total length of the string representation of a modulus
Mirror all patterns *	String	Starting from the first eight bits, checking if the first x bits are present on the remaining string of a modulus (regardless of their position); checked in the same order (left to right)
Mirror all length	String	Length of the largest Mirror all pattern
Mirror all positions *	String	Index positions in which Mirror all pattern is present within each string representation of a modulus; represented in a Python list format
Mirror all percentage	String	It is the percentage that Mirror all pos * Mirror all len represents vs the total length of the string representation of a modulus
Cryptographic library	Target Class	OpenSSH, OpenSSL, GnuTLS, GPG
Library version	Target Class	Version associated with the implementation library

Asterisks (*) indicate the top 14 features selected for key source attribution (generated keys).

4.5 Keys Classification

In this thesis, we explored the performance of six machine learning algorithms commonly used in classification tasks due to their high accuracy and precision: Gaussian Naive Bayes, Neural Network, Decision Trees, Discriminant Analysis, Random Forest, and Logistic Regression analysis. Below, we provide a summary of their characteristics and assumptions:

- **Gaussian Naïve Bayes** (GNB) [9]. It is the most straightforward classification algorithm in Machine Learning classification. It uses the Bayes' theorem, which assumes that a set of features are independent of each other even though this assumption does not typically hold in every case. Regardless, the GNB performs well in real-world scenarios, is faster than other complex algorithms, and requires small training sets to precisely estimate the correct parameters. The model assumes that the data follows a Gaussian distribution (i.e., all data points have continuous values); this assumption holds in both the generated and collected SSH keys.

- **Neural Networks** (NN) [44]. These are a set of algorithms that simulate the human brain to identify relationships within high volumes of complex-non-linear data by implementing non-linear layers and parameters. This model performs well in large datasets. These assumptions hold for the generated SSH keys.

- **Decision Trees** (DT) [11]. It is a discriminative model that produces a sequence of rules for classification; sometimes, it can be unstable due to minimal data points variations. The final classification may be wrong due to these variations as the model could produce different decision trees; moreover, the model can overfit when trying to obtain high purity, thus rendering the classification useless.

- **Discriminant Analysis** (DA). DA comprises discriminant functions based on linear combinations of the features and assumes that each class derives from different Gaussian distributions with the same covariance matrix [38]. During the training phase, the model derives the parameters for each class from its Gaussian distributions. The models generate the functions from a set of data points with known labels; then, it extrapolates these functions to classify data points without labels. Quadratic and Linear Discriminant Analysis are two types of this approach. Linear Discriminant Analysis (LDA) provides linear decision boundaries that make it a good classifier, for instance, in source code attribution authorship.

- **Random Forest** (RF) [30]. RF Performs classification based on several decision trees created from subsets from a dataset and uses their average to improve classification results. When splitting a node through trees' creation, the chosen split is not the best anymore; instead, the split selected is the best split among a random subset of all the features. The result of this randomness increases the forest's bias. However, the variance decreases due to the averaging, which compensates for the increased bias and produces a better model.

- **Logistic Regression** (LR) [17]. LR is a linear classifier that performs predictions based on probabilities; it works well when working with categorical outcomes. However, this classifier requires independent data points since it classifies based on a set of independent variables. This model works well to calculate the probability of an SSH key belonging to a library.

We implemented all these approaches using Python v3.8.5 combined with the Scikit-Learn v0.23.1 and measured their accuracy by using 5-fold cross-validation. We provide a summary of the classification algorithms' parameters we used in Table 4.3.

Table 4.3: Machine learning model parameters

Name	**Parameter**	**Kernel**
Gaussian Naïve Bayes	var_smoothing = 1e-9	Non-linear
Neural Network	max_iter=10000, learning_rate='adaptive', solver='adam', alpha=1	Non-linear
Decision Trees	max_depth=100	Non-linear
Linear Discriminant Analysis	solver = 'svd', shrinkage = None	Linear
Random Forest	n_estimators = 100, min_samples_split = 2, min_samples_leaf = 1, max_features="log2", criterion = 'entropy'	Non-linear
Logistic Regression	penalty="l2", max_iter=100000, solver="lbfgs", multi_class="multinomial"	Linear

5 Validation study

In this chapter, we describe the process we took to generate cryptographic keys and to establish ground data. We detail the source of our generated dataset and describe the main features we used for the attribution process. Also, we briefly compare our results with those obtained by Nemec et al. [47].

5.1 Generated RSA keys

To establish ground truth, we generated 6.5 million RSA keys using four cryptographic libraries on nine Linux distributions. We tested the major versions released for various Linux distributions over the past eight years to explore variations in the generated keys.

To avoid the bias affecting random key generators, we used both the TrueRNG and the NeuG Hardware Random Number Generators to have a steady stream of random numbers through a USB CDC serial port. These tools are ideal for security-related applications, such as generating keys through cryptographic libraries. We used the data generated with these tools to fill our selected operating system's entropy pool, except for the keys generated on Ubuntu, where we used the Haveged tool. We provide the details of the generated keys in Table 5.1. All generated keys are 2048-bit long, as this is the minimum key size recommended by NIST when using the RSA algorithm.

5.2 Validation results

We show the results of attributing 6,767,078 keys to the individual libraries in Figure 5.1 (a). The Random Forest, Logistic Regression, and Neural Network algorithms achieved the best average classification with a 75% accuracy.

As we anticipated, not all the characteristics extracted from the keys contribute equally to the classification accuracy. To ensure that we keep only features with a measurable impact on the overall accuracy, we have retained features with an *Information Gain* of at least 0.005.

Compared to the results when using all our extracted features, we maintained the same accuracy level by only selecting the 14 features that contribute the most to the classification process (Table 4.2). Further analysis of these most significant features revealed that while most features are ranked differently for different libraries, the core features remain the same across libraries and versions (Table 5.2).

One of the challenges that previous studies faced is defining cryptographic rules; for instance, weak

Table 5.1: The summary of the generated keys

OS	OS Version	Year	Type of RNG	OpenSSH library	GnuTLS library	GPG library	OpenSSL library
Ubuntu	20.04	2020	SW (Haveged)	8.2p1	3.6.13	2.2.19	1.1.1d
Ubuntu	18.04	2018	SW (Haveged)	7.6p1	3.5.18	2.2.4	1.1.1
Ubuntu	16.04	2016	SW (Haveged)	7.2p2	3.4.10	2.1.11	1.0.2g
Ubuntu	14.04	2014	SW (Haveged)	6.6	*	2.0.22	1.0.1f
Ubuntu	12.04	2012	SW (Haveged)	5.9	*	*	*
Mint	20	2020	HW (RNG)	8.2p1	3.6.13	2.2.19	1.1.1f
Mint	19	2020	HW (RNG)	7.6p1	3.5.18	2.2.4	1.1.1h
Fedora	30	2019	HW (RNG)	8.0p1	3.6.10	2.2.13	1.1.1b
Fedora	23	2015	HW (RNG)	7.1p1	3.4.5	2.1.7	1.0.2d
Fedora	20	2014	HW (RNG)	6.3	3.1.16	*	1.0.1e
Fedora	17	2012	HW (RNG)	5.9	*	*	1.0.0i
Fedora	14	2010	HW (RNG)	5.5	*	*	1.0.0a
CentOS	8.2.2004	2019	HW (RNG)	8.0p1	3.6.8	*	1.1.1c
Manjaro	20	2020	HW (RNG)	8.3p1	3.6.15	2.2.23	1.1.1g
Swift	4.19.0	2018	HW (RNG)	7.9p1	*	*	1.1.1d
Endeavour	5.8	2020	HW (RNG)	8.3p1	3.6.15	2.2.23	1.1.1g
Kali	2020.3	2020	HW (RNG)	8.3p1	3.6.15	2.2.20	1.1.1g
Oracle	R8	2019	HW (RNG)	8.0p1	3.6.8	*	1.1.1c
Oracle	R7	2017	HW (RNG)	7.4p1	*	*	1.0.2k
Oracle	R6	2013	HW (RNG)	5.3p1	*	*	1.0.1e
Total generated keys				3,084,936	1,165,984	616,159	1,899,999

Asterisks (*) indicate cases when certain libraries are no longer available for specific Linux distributions.

Table 5.2: The list of core features across for various libraries

Brotli compression, LRS pattern, LRS position, Zeroes position, Ones position, Mirror all positions, Mirror position, Mirror all patterns, Bin

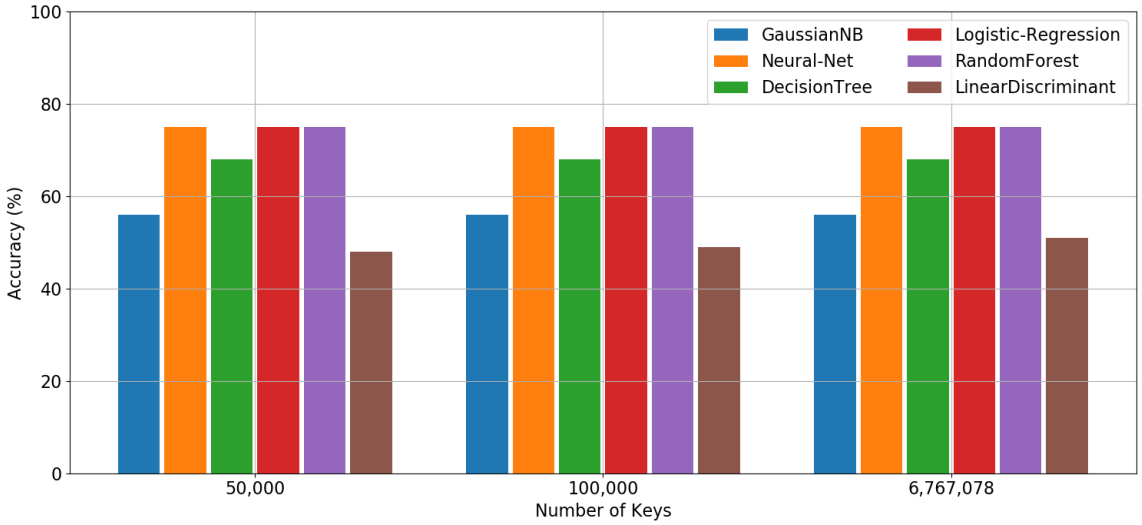
implementation decisions effectively leading to bias in the produced keys. The features that we show in Table 5.2 are independent of the underlying libraries.

While classifiers' performance remains somewhat stable for a different number of keys, we found that accuracy levels vary depending on the source library's characteristics.

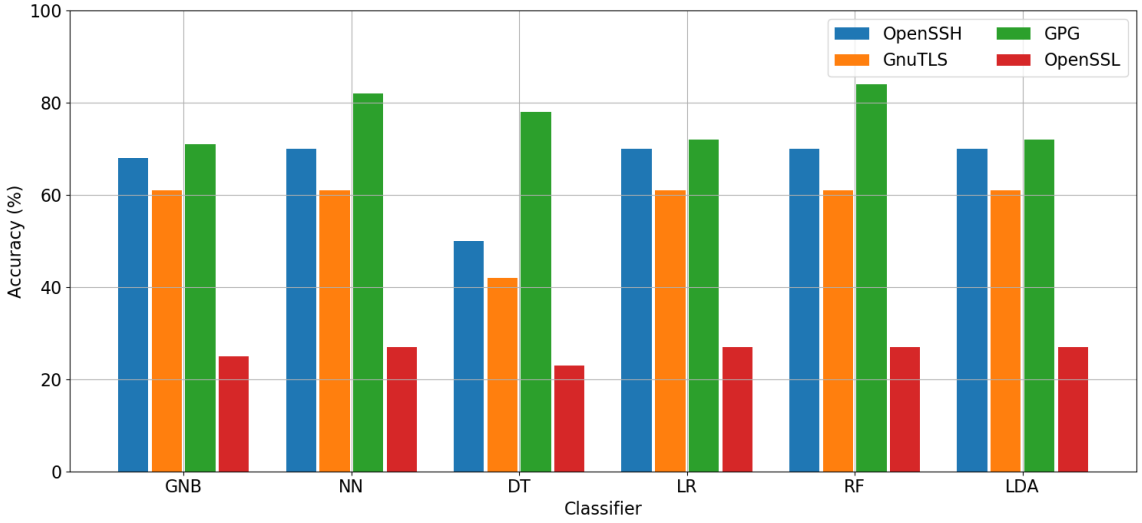
To further understand our approach's granularity to discriminate keys, we evaluated attribution across library versions. Figure 5.1 (b) shows the average accuracy of attributing the individual keys to their libraries' versions. In this experiment, the Random Forest classification algorithm obtained an 85% average accuracy. Since both experiments showed that the Random Forest classification algorithm performs the best in our settings, we further use this classifier in our analysis.

Table 5.3 presents more granular results of attributing keys by major and minor release versions with their corresponding libraries[1]. Since we do not have enough information about keys for the GnuTLS, GPG, and OpenSSL libraries to distinguish between major versions, we experimented only with their major and

[1]We refer to library versions using the conventional notation of software versioning represented by major.minor[.build[.patch]]



(a) Attribution of keys to their source libraries



(b) Inter-Library attribution of keys to the corresponding library version

Figure 5.1: The accuracy of attributing generated keys to the originating library and library versions

Table 5.3: Attribution accuracy for library versions (Random Forest)

Library	Version	Accuracy	Number of keys
OpenSSH	5.x	100%	391,928
OpenSSH	6.x	0%	193,009
OpenSSH	7.x	24%	700,000
OpenSSH	8.x	63%	1,799,999
GnuTLS	3.1.x	0%	100,000
GnuTLS	3.4.x	0%	200,000
GnuTLS	3.5.x	0%	154,142
GnuTLS	3.6.x	61%	711,842
GPG	2.0.x	76%	28,657
GPG	2.1.x	98%	149,010
GPG	2.2.x	82%	438,492
OpenSSL	1.0.x	53%	800,000
OpenSSL	1.1.x	63%	1,099,999

minor releases.

We can observe that specific libraries have distinguishable patterns (e.g., GPG, OpenSSH 8.x). Theoretically, code changes applied to a library that occurred within a major version should not significantly impact the generated keys. On the other hand, significant changes that culminate with releasing a new version should equate to marked differences. Despite our assumptions, our results seem to suggest that, regardless of the library type or version, it is possible to attribute a cryptographic key not only to the library type but, more specifically, to its specific major and minor versions.

The GPG library is the one that differs the most among versions; we could attribute keys to specific versions with an accuracy ranging from a minimum of 76% for the 2.0.x version to a maximum of 98% for the 2.1.x version.

The accuracy of other types of libraries varies depending on the library version. For example, we could only attribute keys to GnuTLS 3.6 with a 61% accuracy. Our current assumption is that such variability may arise from changes in the libraries' internal logic or structure of cryptographic primitives implemented by a library.

To further understand this phenomenon, we analyzed the release notes for each library (GnuTLS[2], OpenSSH[3], GPG[4], OpenSSL[5] and aggregated the library versions available in our generated set to reflect the changes in libraries that involve modifications (i.e., improvements) of the random number generation pro-

[2]https://gitlab.com/gnutls/gnutls/blob/master/NEWS
[3]https://www.openssh.com/releasenotes.html
[4]https://gnupg.org/download/release_notes.html
[5]https://www.openssl.org/news/changelog.html

cess. As a side note, the GPG release notes did not provide a sufficient level of details on such modifications; therefore, we could not derive further groupings for this library.

Table 5.4 gives an insight into the variability in the attribution of keys when we only consider the libraries' major and minor versions. For example, the OpenSSH 5.9/5.9p1 release switched to obtaining random numbers directly from OpenSSL or from a PRNGd/EGD instance specified at configuration time. This modification caused a structural change in produced keys resulting in distinctive patterns in keys generated before the 5.9 release. Hence, we could distinguish keys generated with earlier versions with an 81% accuracy.

We also note that, in some cases, drastic changes to underlying libraries happen even within build releases. For example, OpenSSL 1.1.1d came out with a completely rewritten random number generator, which resulted in 93% of the produced keys positively linked to this specific version of OpenSSL.

In some cases (e.g., GnuTLS before version 3.5.19 and OpenSSH between version 6.8 and 7.8), our approach was unsuccessful in attributing keys to their corresponding library, thus achieving a 0% accuracy. This percentage is a general expectation of cryptographic keys; the produced key should not bear any signs of the originating library.

We speculate that changes involving the memory usage of the randomness pools and the decision to leave to the operating system the responsibility to ensure a proper initialization during early boot time negatively impact the entropy distillation process that leads to the presence of discernable patterns in the resulting cryptographic key.

5.2.1 Comparative analysis

To better estimate our solution's accuracy and efficacy, we decided to compare its performance to the model's performance developed by Nemec et. al [47]. We have, therefore, implemented and applied their approach to the set of our generated keys. We give these results in Table 5.5.

Nemec et al. [47] approach showed the feasibility of attributing keys to groups of similar libraries (with over 94% accuracy). Nevertheless, it fails to trace individual keys to their corresponding libraries, achieving at most 42% accuracy. Since their accuracy is low, we did not evaluate a more granular attribution to specific library versions.

Table 5.4: Attribution accuracy of generated keys by minor build version groupings (Random Forest)

Library	**Release groupings**	**Accuracy**	**Number of keys**
OpenSSH	[5.3p1 - 5.9]	**81%**	191,928
OpenSSH	[6.3]	50%	293,009
OpenSSH	[6.6]	0%	100,000
OpenSSH	[7.1p1 - 7.2]	0%	200,000
OpenSSH	(7.2 - 7.6]	0%	300,000
OpenSSH	(7.9+]	**69%**	199,999
GnuTLS	[3.1.16]	0%	100,000
GnuTLS	[3.4.5 - 3.4.10]	0%	200,000
GnuTLS	[3.5.18]	0%	154,142
GnuTLS	[3.6.8+]	**61%**	711,842
GPG	2.0.x	**76%**	28,657
GPG	2.1.x	**98%**	149,010
GPG	2.2.x	**82%**	438,492
OpenSSL	(1.0.0a - 1.0.1f]	13%	200,000
OpenSSL	(1.0.1f - 1.0.2k]	20%	300,000
OpenSSL	(1.0.2k - 1.1.0]	20%	300,000
OpenSSL	[1.1.1]	34%	299,999
OpenSSL	[1.1.1b]	8%	100,000
OpenSSL	[1.1.1c]	13%	200,000
OpenSSL	[1.1.1d]	**93%**	200,000
OpenSSL	(1.1.1d - 1.1.1h]	34%	300,000

* An inclusive bound is represented by '[', an exclusive bound is represented by '('.
** Based on the available data in Table 5.1.

Table 5.5: Classification accuracy of Nemec et. al [47] approach on the generated keys

Model	**Accuracy**
GaussianNB	35%
Neural-Net	42%
Decision Tree	23%
Logistic Regression	42%
Random Forest	40%
Linear Discriminant	42%

6 Internet wide scan of Canadian IPv4 space

In this chapter we provide a full explanation on how we collected, parsed and grouped the datasets for our research. We also elaborate on their most important characteristics and show statistics that helped us in our analysis.

6.1 The collected SSH dataset

For our analysis, we collected a list of 45 million IPv4 addresses (IPs)[1]. We filtered the IPs based on their legal owner registration resulting in 222,829 publicly facing Canadian IPs accepting connections on either port 22 or 2222, or both; from these IPs, we identified 220,837 unique IPs and 1992 repeated IPs. The repeated IPs are most likely attributed to hosts accepting connections on both ports and serving two different SSH keys. We continuously scanned these hosts for 83 days from August to November 2019 and collected keys using the Secure Shell Host (SSH) protocol. Over this period, we gathered a total of 191,976 SSH keys; from these, 191,005 retrieved through SSH v2.0 protocol (Table 6.1).

From the collected cryptographic keys, we identified 155,107 using *OpenSSH* library, which is known to be the most widely used on the Internet today. Moreover, we also identified a few SSH keys using older versions of the library (e.g., OpenSSH 1.x and OpenSSH 3.x were released in 2000 and 2001, respectively); it is worth noting that we did not find any key generated with OpenSSH 2.x. We also noticed 89 keys generated with OpenSSH, but we could not identify the associated version. Moreover, we found 89 keys generated by *OpenSSH version 12*, **which does not exist**; the newest version is 8.x.

We extracted the RSA public keys' components (i.e., modulus and exponent) from the SSH certificates and compiled a set of 110,798 unique moduli along with seven unique exponents. We then organized the SSH keys according to their size (i.e., the moduli size) and identified 24,400 keys with a *legacy* or *deprecated* status, we refer to these as having a *non-acceptable* status.

It is important to note that NIST compliant RSA keys are required to have lengths greater or equal to 2048-bits [7], while the US National Security Agency and the US Central Security Service (NSA/CSS) compliant RSA keys are required to have lengths greater or equal to 3072-bits [46]. The latter only accepts lengths equal or greater to 2048-bits in already deployed Public Key Infrastructure systems transitioning from SHA-1 to SHA-256 [48]. We found 167,576 keys with a size equal or above 2048-bits, thus in an *acceptable* status.

[1] We extracted these IP addresses from the www.IP2location.com service on 28 July, 2019

Total distinct IPv4 hosts scanned	220,837
Distinct SSH RSA keys	191,976
Distinct SSH RSA moduli	110,798
Distinct SSH RSA exponents	7
Keys with moduli size $\geq$ to 2048 bits	167,576
Key with moduli size $<$ 2048 bits	24,400
SSH version 1.99	971
SSH version 2.0	191,005
OpenSSH library	155,107
Other library	36869

Table 6.1: General statistics of the retrieved keys

6.2 Parsing

To retrieve the RSA keys from the SSH certificates, we automatized the decoding process. We show the structure of the certificates and the decoding steps in Figure 6.1. Also, we note that all derived RSA keys come from the downloaded SSH certificate files that follow the format shown in Figure 6.1.

The actual RSA key (i.e., modulus and exponent) is within the 'ssh-rsa rsa_public_key_base64' part. Once extracted, we wrote the RSA public keys into PUB files, which we decoded following the parsing procedure described in Section 4.2.

In addition to the RSA key, we also extracted the file's creation date, the IP address, port associated with the SSH key, protoversion, library, associated comment if it exists, the library name and version used to generate the SSH keys, and the presence or absence of the High-Performance SSH/SCP patch. Finally, we obtained the SSH key's SHA256 and MD5 fingerprints. This additional information can be potentially complementary to the numerical and textual features, and thus we explored its contribution to our source attribution analysis. We refer to these data points as protocol-related features.

6.3 Security analysis of the SSH collected keys

We started our analysis by checking if the collected exponents (e) are in compliance with the NIST recommendations found in [7]. To claim compliance with these specifications, all RSA implementations are required to select an *odd* integer e such that $65537 \leq e < 2^{256}$ before the generation of p and q. Since we did not perform any code inspection on RSA implementations, we cannot check if the implementations select value for all e before generating the prime factors p and q. For simplicity and without loss of generality, we considered values for e that are between the expected numerical range and are also odd as being *in compliance*

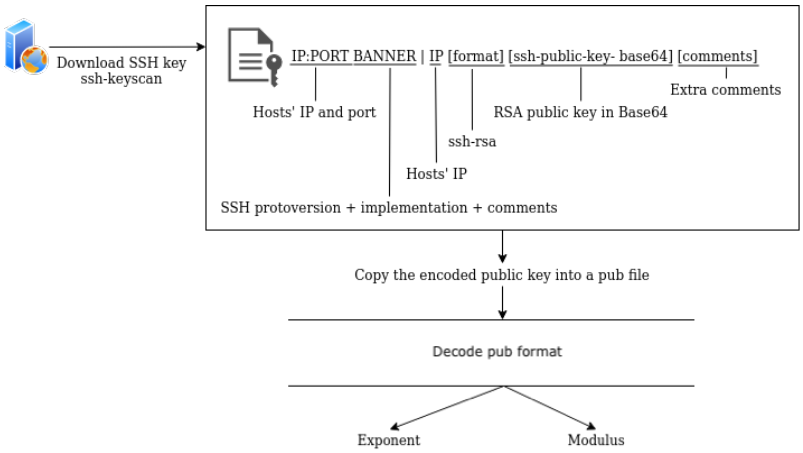


Figure 6.1: The decoding process of the collected SSH keys.

with NIST specifications. We first checked if the values of e in our dataset were odd integers and found that, indeed, they all were odd positive integers. Next, we grouped exponents by value and found that 27,557 of them are *not in compliance* with NIST specifications as they have values for $e < 65337$.

6.3.1 Low public exponents

Implementations select Small values for RSA public exponents (Small values for RSA public exponents (e) due to the need for faster and inexpensive computational operations when encrypting messages (m). According to the NIST recommendations, any value of (e) lower than 65,537 is considered small and consequently results in a weak key. Specifically:

- **If $e = 1$:** Using $e = 1$ poses a serious security threat since it allows the computation of the $Priv_k$ from the Pub_k due to the relationship between N, e and d while at the same time leaving any m in the clear (i.e., $ciphertext = m$). We did not find any instance from our datasets in which $e = 1$.

- $1 < e < 65,537$: these values could be potentially problematic in the absence of a proper padding scheme. As stated above, we discovered 27,557 exponents within this group. It has been shown that small e values allow attackers to find m from e number of ciphertexts in the absence of a padding scheme. It is also possible to find d from $\frac{e(e+1)}{2}$ number of ciphertexts when messages are linearly related to each other (e.g., by using a timestamp for padding) [27]. Moreover, attackers can compute m if there exists a polynomial relationship between at least two messages in conjunction with a small value of e and the reuse of N [15]. As we will see later, we did find RSA keys with small e and reuse of moduli.

Test	Result
Birthdays Test	failed
Overlapping 5-Permutations Test	failed
32x32 Binary Rank Test	failed
6x8 Binary Rank Test	failed
Bitstream Test	failed
Diehard OPSO	failed
Diehard OQSO Test	failed
Diehard DNA Test	failed
Diehard Count the 1s (stream) Test	failed
Diehard Count the 1s Test (byte)	failed
Diehard Parking Lot Test	failed
Diehard Minimum Distance (2d Spheres) Test	passed
Diehard 3d Sphere (Minimum Distance) Test	passed
Diehard Squeeze Test	failed
Diehard Sums Test	failed
Diehard Runs Test	failed
Diehard Craps Test	failed
Marsaglia and Tsang GCD Test	failed
STS Monobit Test	failed
STS Runs Test	failed
STS Serial Test (Generalized)	failed
RGB Bit Distribution Test	failed
RGB Generalized Minimum Distance Test	passed
RGB Permutations Test	failed
RGB Lagged Sum Test	failed
RGB Kolmogorov-Smirnov Test	failed
Byte Distribution	failed
DAB DCT	failed
DAB Fill Tree Test	failed
DAB Fill Tree 2 Test	failed
DAB Monobit 2 Test	failed

Table 6.2: The results of the Dieharder test on the collected SSH keys

6.4 Analysis of collected key moduli

We first checked the moduli uniqueness and found that 110,798 were indeed unique, while 81,178 were shared. Sharing moduli could allow an attacker to find m from C_i if encryption happened with distinct relatively prime e but a shared N [55]. Moreover, Delaurentis et al. [18] showed that when users share the same N, an attacker can decrypt an arbitrary number of m from any user or forge digital signatures without the need of factorizing N.

We grouped the moduli according to their size and discovered 24,400 moduli with a *non-acceptable* status due to them having sizes smaller to what NIST recommends. Moreover, NIST recommendations also require moduli sizes to be even; in this regard, we checked if the collected moduli had a proper format, and we found 10,158 moduli with odd sizes. In Figure 6.2, we show the breakdown of moduli per sizes.

6.4.1 Moduli distribution

We would expect the resulting modulus to be random since each RSA modulus results from two random primes. Randomly chosen numbers follow a uniform distribution; this implies that moduli distributions must also fit a uniform distribution regardless of their origin and size. To corroborate this property, we first divided

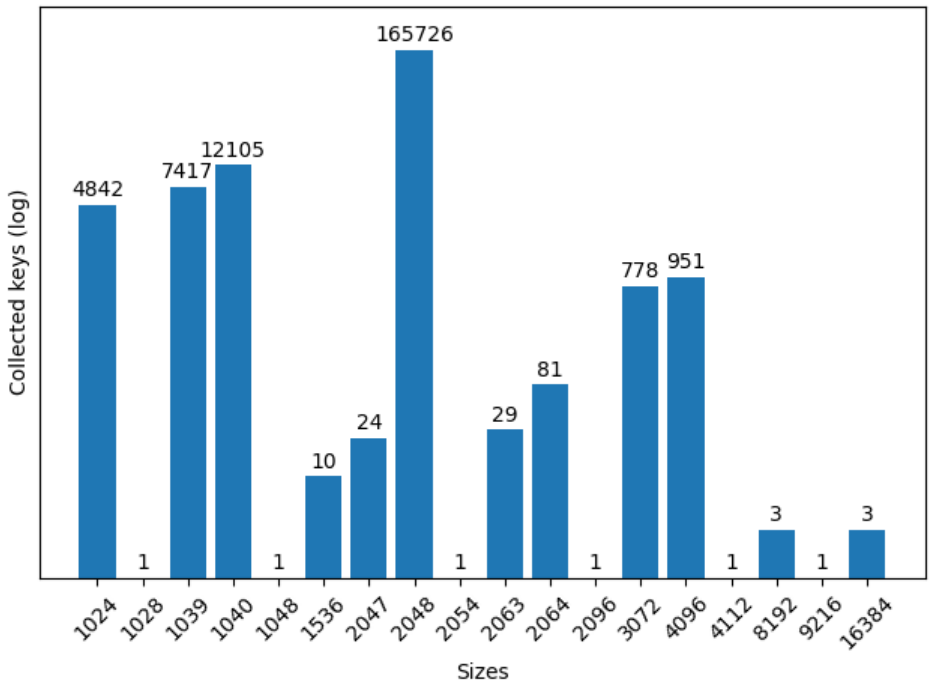


Figure 6.2: Collected RSA moduli per size

the moduli into groups according to their size; then, we sorted moduli values by created bins ranging from one to one-hundred, representing the lowest and the highest value, respectively. Then, we plotted them to have a visual representation of their distribution; we show the distributions for all the collected keys in Figure 6.3. It is clear that none of the groups fit into a uniform distribution; this suggests that neither libraries nor random number generators are choosing primes p and q randomly due to fix patterns or insufficient entropy.

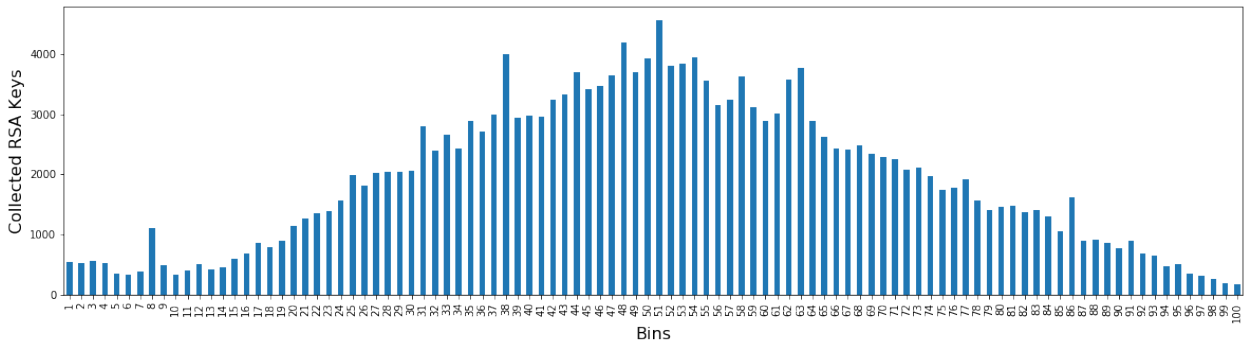


Figure 6.3: Distribution of all collected RSA keys

Table 6.3: The list of top features (Random Forest)

Features	Description	Accuracy
Top 1	Mirror all patterns	62%
Top 2	Bin, Offset	65%
Top 5	Bin, LZMA compression, Offset, Brotli compression, Mirror all patterns	78%

6.5 Source Attribution

To measure our attribution's performance, we designed a set of experiments for examining the effect of the key features on the accuracy of different types of classifiers.

To analyze the importance of dataset size, we randomly selected eight subsets of 191,976 keys collected from the Internet and created sub-samples with a size of 1000, 10000, 50000, 100,000 and 191,975 keys each. For each of the subsets, we allocated 75% of data for training purposes and 25% for testing, using a random sampling technique that uses a stratified k-folding to reduce bias while increasing the likelihood of having balanced samples.

Subsequently, we performed two experiments; the fist to identify the type of library regardless of the major or minor versions and the second one, where we tried to associate each key with a major and minor version positively. We performed such experiments with different combinations of feature sets to understand what type of features may provide a better source attribution.

Figure 6.4 shows the ability of six distinct types of classifiers to attribute a key to a specific library, according to different combinations of feature sets.

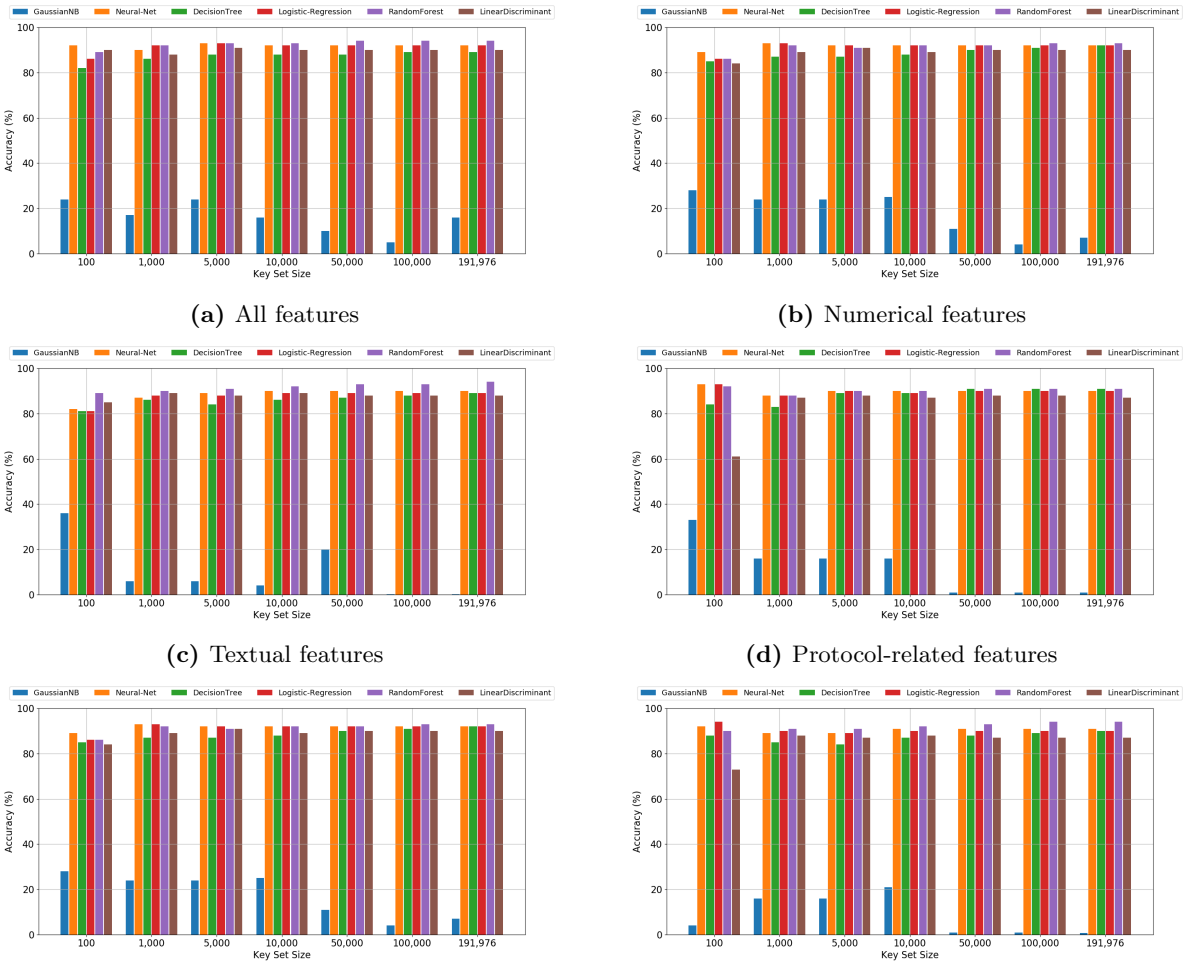
Most classifiers have comparable performance, except for Gaussian Naive Bayes. We can reach over 90% accuracy with Random Forest across all sets of features for 191,976 keys with our approach. When using all, textual alone and protocol-related together with textual features combined with the Random Forest classifier (Figure 6.4 (a), (c) and (f)), we achieved 95% accuracy; the best accuracy among all set of features.

It is not unexpected to see the effect of protocol-related features on attribution; nevertheless, we note that textual features that only retain a modulus's internal characteristics provide the same accuracy.

This finding supports our earlier assumption that the logic or structure of cryptographic primitives implemented in a library leaves distinct traces on the generated key modulus.

Table 6.3 clarifies the impact that certain features may have on accuracy. We can reduce the features set of the top 14 features selected in Section 5 to six while retaining a significant portion of our original accuracy (i.e., 78% with only five modulus features). We describe these top features in Table 6.3.

If we focus our attention on the size of the data we used, our analysis reveals that most classifiers perform consistently well across different sample sizes that have at least with 1000 keys (Figure 6.4), but seems to



(a) All features

(b) Numerical features

(c) Textual features

(d) Protocol-related features

(e) Protocol-related and numerical features

(f) Protocol-related and textual features

Figure 6.4: Attribution accuracy of collected keys by individual library

perform poorly with sample sizes that have less then 1000 keys; this results in significant loss of accuracy that is proportional to the size of the sample.

We give the results of the more granular experiments (i.e., attribution to a source library and its version) in Table 6.4. In most cases, we can attribute keys to their originating library and its version. The performances range between 68% to 100% attribution accuracy. The performance is generally lower for libraries/versions with fewer keys, which we expected due to insufficient data.

It is interesting that although we were not able to obtain any information for 'Unknown' libraries or versions, they seem to represent homogeneous groups with distinct patterns. For example, we achieved 89% accuracy in classifying an Unknown library with an Unknown version. Similarly, the non-existent 12.x version of the OpenSSH library can be accurately traced, with a 100% accuracy, to individual keys.

Table 6.4: Attribution results for the collected SSH keys by the individual library and its version (Random Forest)

Library	**Version**	**Accuracy**	**Number of keys**
Dropbear	0.x	100%	83
Dropbear	2011.x	100%	222
Dropbear	2012.x	87%	16,521
Dropbear	2013.x	100%	220
Dropbear	2014.x	98%	1,212
Dropbear	2015.x	98%	538
Dropbear	2016.x	98%	2,854
Dropbear	2017.x	100%	236
Dropbear	2018.x	91%	143
Dropbear	2019.x	100%	146
Dropbear	Unknown	68%	5,776
FTP Server	3.x	0%	3
NetVanta	4.x	0%	2
OpenSSH	3.x	78%	503
OpenSSH	4.x	99%	4,484
OpenSSH	5.x	91%	23,279
OpenSSH	6.x	98%	25,564
OpenSSH	7.x	79%	99,564
OpenSSH	8.x	82%	1,535
OpenSSH	12.x	100%	89
OpenSSH	Unknown	100%	89
Reflection	7.x	0%	1
iLO	0.x	100%	34
Unknown	Unknown	89%	8,787

As announced in their SSH certificate.

7 Conclusion

Assessing RSA keys' characteristics through Machine Learning techniques allowed us to identify a series of potentially significant aspects directly linked to the level of RSA keys' security.

The general expectation of cryptographic keys requires that all produced keys do not bear any artifact from the originating library. Still, we have positively associated a percentage of the collected RSA keys to a specific library and more worryingly to specific library versions.

We generated 6,767,078 keys with different cryptographic libraries, different versions and on different platforms. We attributed the keys moduli to libraries with an average accuracy of 75% and their versions with an average accuracy of 85%.

Our results highlight that some core logic in the cryptographic libraries may be generating keys that do not have the anticipated level of security.

We found features that can discriminate between keys generated by different libraries, between keys with a different major, minor, build and in some cases, with different patch versions. For example, we obtained 76% for GPG 2.0.x and 98% for GPG 2.1.x.

In the second round of experiments, we collected 191,976 SSH keys from the Internet, and we used six classifiers to analyze them. Our results show that our approach can attribute individual keys to specific library versions with accuracy levels between 68% to 100%. We achieved this without any prior knowledge of the system or the library that generated them.

7.1 Future work

This research work unravels a whole new set of exciting research paths involving RSA cryptographic keys and their use in critical real-life applications. Consequently, several research projects could be pursued based on the presented approach:

- **Attribute the source of cryptocurrency addresses.** Cryptocurrencies provide a high degree of privacy and anonymity to users by implementing hashed public keys as addresses to relay or receive payments. Hashing algorithms are irreversible by design; however, if the public keys feeding the hashing algorithms or the hashing algorithms themselves were to suffer similar deficiencies as found in this research (e.g., distinguishable patterns), the resulting hashes could be used to attribute the wallet, system or library behind the addresses.

- **Extend the current approach to cover proprietary libraries implementing the RSA algorithm, such as Microsoft SChannel, WolfSSL.** OpenSSH is indeed the most used implementation found in the wild due to its Open Source nature. Nonetheless, the approach presented in this paper is platform and library-independent; thus, it can work across a wide range of implementations and systems. Extending this approach to several black-boxed proprietary libraries could provide an insight into their security levels.

- **Analyze the security characteristics of RSA keys associated with TOR hidden services.** Users' privacy and security while accessing the TOR network depend, in part, on the cryptographic keys associated with the hidden service they are visiting. In this regard, this thesis's security concerns could endanger users and the TOR network since it could provide an accurate way to attribute the operating system behind the hidden services; thus, opening the door for interested parties to engage in targeted cyberattacks.

- **Analyze the security of RSA keys associated with Windows' signed applications.** Validating software developers has many security-related implications. On the one hand, it allows users to make informed decisions about the trustworthiness of specific applications. On the other hand, insecure implementations could lead to the impersonation of entities and the forgery of digital signatures; thus, weakening the trust placed on the digital signature system.

References

[1] OpenSSH. Available online: https://www.openssh.com/. Last accessed: 2020-11-06.

[2] OpenSSL. Available online: https://www.openssl.org/. Last accessed: 2020-11-06.

[3] The Global Internet Phenomena Report. Technical Report October, Sandvine, 2018. Available online: https://www.sandvine.com/hubfs/downloads/phenomena/2018-phenomena-report.pdf. Last accessed: 2020-11-06.

[4] Mustafa Emre Acer, Emily Stark, Adrienne Porter Felt, Sascha Fahl, Radhika Bhargava, Bhanu Dev, Matt Braithwaite, Ryan Sleevi, and Parisa Tabriz. Where the wild warnings are: Root causes of chrome https certificate errors. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1407–1420, New York, NY, USA, 2017. ACM.

[5] Ross Anderson. Why cryptosystems fail. In *Proceedings of the 1st ACM conference on Computer and communications security - CCS '93*, pages 215–227, New York, New York, USA, 1993. ACM Press.

[6] Robert Baldwin, Clyde Monma, George Cox, Eric Murray, Cheri Dowell, Avi Rubin, Stuart Haber, Don Stephenson, Burt Kaliski, and Joe Tardo. The Secure Sockets Layer (SSL) Protocol Version 3.0, 2011. Available online: https://tools.ietf.org/pdf/rfc4648.pdf. Last accessed: 2020-11-06.

[7] Elaine Barker, Lily Chen, Allen Roginsky, Apostol Vassilev, Richard Davis, and Scott Simon. Recommendation for pair-wise key establishment using integer factorization cryptography. Technical report, National Institute of Standards and Technology, Gaithersburg, MD, mar 2019.

[8] Lawrence E. Bassham, Andrew L. Rukhin, Juan Soto, James R. Nechvatal, Miles E. Smid, Elaine B. Barker, Stefan D. Leigh, Mark Levenson, Mark Vangel, David L. Banks, and et al. Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, Gaithersburg, MD, USA, 2010.

[9] Thomas Bayes. Lii. an essay towards solving a problem in the doctrine of chances. by the late rev. mr. bayes, frs communicated by mr. price, in a letter to john canton, amfr s. *Philosophical transactions of the Royal Society of London*, (53):370–418, 1763.

[10] Karthikeyan Bhargavan, Ioana Boureanu, Pierre-Alain Fouque, Cristina Onete, and Benjamin Richard. Content delivery over tls: a cryptographic analysis of keyless ssl. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 1–16, April 2017.

[11] Leo Breiman, Friedman Jerome, Stone Charles J., and Olshen R.A. *Classification and regression trees*. Wadsworth International Group, 1984.

[12] Robert G. Brown. DieHarder: A Gnu Public Licensed Random Number Tester. Technical report, 2006.

[13] Robert G. Brown, Dirk Eddelbuettel, and David Bauer. Dieharder. Available online: https://webhome.phy.duke.edu/∼rgb/General/dieharder.php. Last accessed: 2020-11-06.

[14] Jon Callas, Lutz Donnerhacke, Hal Finney, David Shaw, and Rodney Thayer. OpenPGP message format, 2007. Available online: https://tools.ietf.org/pdf/rfc4880.pdf. Last accessed: 2020-11-06.

[15] Don Coppersmith, Matthew Franklin, Jacques Patarin, and Michael Reiter. Low-Exponent RSA with Related Messages. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 1070, pages 1–9. 1996.

[16] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 95–110, Berkeley, CA, USA, 2014. USENIX Association.

[17] David Roxbee Cox and E Joyce Snell. *Analysis of binary data*, volume 32. CRC press, 1989.

[18] John M. Delaurentis. A Further Weakness in the Common Modulus Protocol for the RSA Cryptoalgorithm. *Cryptologia*, 8(3):253–259, jul 1984.

[19] Whitfield Diffie and Martin E. Hellman. Multiuser cryptographic techniques. In *Proceedings of the June 7-10, 1976, national computer conference and exposition on - AFIPS '76*, pages 109–112, New York, New York, USA, 1976. ACM Press.

[20] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, nov 1976.

[21] Zakir Durumeric, James Kasten, Michael Bailey, and J. Alex Halderman. Analysis of the https certificate ecosystem. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 291–304, New York, NY, USA, 2013. ACM.

[22] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. ACM.

[23] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications Security*, CCS '13, page 73–84, New York, NY, USA, 2013. Association for Computing Machinery.

[24] Adam Everspaugh, Yan Zhai, Robert Jellinek, Thomas Ristenpart, and Michael Swift. Not-so-random numbers in virtualized linux and the whirlwind rng. In *2014 IEEE Symposium on Security and Privacy*, pages 559–574, May 2014.

[25] Joseph Galbraith and Rodney Thayer. The Secure Shell (SSH) Public Key File Format, 2006. Available online: https://tools.ietf.org/pdf/rfc4716.pdf. Last accessed: 2020-11-06.

[26] Oliver Gasser, Ralph Holz, and Georg Carle. A deeper understanding of ssh: Results from internet-wide scans. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–9, May 2014.

[27] Johan Hastad. On Using RSA with Low Exponent in a Public Key Network. In *Advances in Cryptology — CRYPTO '85 Proceedings*, pages 403–408. Springer Berlin Heidelberg, Berlin, Heidelberg, 1985.

[28] Marcella Hastings, Joshua Fried, and Nadia Heninger. Weak keys remain widespread in network devices. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, pages 49–63, New York, NY, USA, 2016. ACM.

[29] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 205–220, Bellevue, WA, 2012. USENIX.

[30] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE, 1995.

[31] Internet Engineering Task Force (IETF). Brotli Compressed Data Format. Available online: https://tools.ietf.org/html/rfc7932. Last accessed: 2020-11-06.

[32] Adam Janovsky, Matus Nemec, Petr Svenda, Peter Sekan, and Vashek Matyas. Biased RSA Private Keys: Origin Attribution of GCD-Factorable Keys. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 12309 LNCS, pages 505–524. 2020.

[33] Adam Janovsky, Matus Nemec, Petr Svenda, Peter Sekan, and Vashek Matyas. Biased rsa private keys: Origin attribution of gcd-factorable keys. In *European Symposium on Research in Computer Security*, pages 505–524. Springer, 2020.

[34] Simon Josefsson and Sean Leonard. Textual Encodings of PKIX, PKCS, and CMS Structures, 2015.

[35] Werner Koch, David Shaw, Niibe Yukata, Jussi Kivilinna, and Andre Heinecke. The gnu privacy guard, 2020. Available online: https://gnupg.org/people/index.html. Last accessed: 2020-11-06.

[36] Geir M. Køien. Why Cryptosystems Fail Revisited. *Wireless Personal Communications*, 106(1):85–117, may 2019.

[37] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-Hashing for Message Authentication, 1981. Available online: https://tools.ietf.org/pdf/rfc2104.pdf. Last accessed: 2020-11-06.

[38] Peter A Lachenbruch and M Goldstein. Discriminant analysis. *Biometrics*, pages 69–85, 1979.

[39] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why does cryptographic software fail? In *Proceedings of 5th Asia-Pacific Workshop on Systems - APSys '14*, pages 1–7, New York, New York, USA, 2014. ACM Press.

[40] Sami Lehtinen and Chris Lonvick. The Secure Shell (SSH) Protocol Assigned Numbers, 2006. Available online: https://tools.ietf.org/pdf/rfc4250.pdf. Last accessed: 2020-11-06.

[41] Juanru Li, Zhiqiang Lin, Juan Caballero, Yuanyuan Zhang, and Dawu Gu. K-hunt: Pinpointing insecure cryptographic keys from execution traces. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 412–425, New York, NY, USA, 2018. ACM.

[42] Yong Li, Yuanyuan Zhang, Juanru Li, and Dawu Gu. iCryptoTracer: Dynamic Analysis on Misuse of Cryptography Functions in iOS Applications. In Man Ho Au, Barbara Carminati, and C.-C. Jay Kuo, editors, *Network and System Security*, pages 349–362, Cham, 2014. Springer International Publishing.

[43] John Linn. Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures, 1993. Available online: https://tools.ietf.org/pdf/rfc1421.pdf. Last accessed: 2020-11-06.

[44] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[45] Ildar Muslukhov, Yazan Boshmaf, and Konstantin Beznosov. Source attribution of cryptographic api misuse in android applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, page 133–146, New York, NY, USA, 2018. Association for Computing Machinery.

[46] National Security Agency and Central Security Service. Commercial National Security Algorithm Suite and Quantum Computing: FAQ. *Information assurance directorate*, 2016.

[47] Matus Nemec, Dusan Klinec, Petr Svenda, Peter Sekan, and Vashek Matyas. Measuring popularity of cryptographic libraries in internet-wide scans. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, pages 162–175, New York, NY, USA, 2017. ACM.

[48] Committee on National Security Systems. Use of Public Standards for the Secure Sharing of Information Among National Security Systems, July,2015. Available online: http://www.cnss.gov/CNSS/openDoc.cfm?yYGcisWfnxXHhsnVAmCb/w==. Last accessed: 2020-11-06.

[49] Luca Piccolboni, Giuseppe Di Guglielmo, Luca P. Carloni, and Simha Sethumadhavan. Crylogger: Detecting crypto misuses dynamically. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2021.

[50] Jon Postel. Transmission Control Protocol, 1981. Available online: https://tools.ietf.org/pdf/rfc0793.pdf. Last accessed: 2020-11-06.

[51] Python. The Lempel–Ziv–Markov chain (LZMA) compression algorithm. Available online: https://docs.python.org/3/library/lzma.html. Last accessed: 2020-11-06.

[52] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2455–2472, New York, NY, USA, 2019. Association for Computing Machinery.

[53] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3, 2018. Available online: https://tools.ietf.org/pdf/rfc8446.pdf. Last accessed: 2020-11-06.

[54] Ronald Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, feb 1978.

[55] Gustavus J. Simmons. A "Weak" Privacy Protocol Using the RSA Cryptoalgorithm. *Cryptologia*, 7(2):180–182, apr 1983.

[56] Petr Svenda, Matus Nemec, Peter Sekan, Rudolf Kvasnovsky, David Formanek, David Komarek, and Vashek Matyas. The million-key question—investigating the origins of RSA public keys. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 893–910, Austin, TX, August 2016. USENIX Association.

[57] Tim Rühsen, Daiki Ueno, Dmitry Baryshkov. The GnuTLS Transport Layer Security Library. Available online: https://gnutls.org/. Last accessed: 2020-11-06.

[58] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Undefined Behavior: What Happened to My Code? In *Proceedings of the Asia-Pacific Workshop on Systems - APSYS '12*, pages 1–7, New York, New York, USA, 2012. ACM Press.

[59] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*, page 26, New York, New York, USA, 2011. ACM Press.

[60] Tatu Ylonen and Chris Lonvick. The Secure Shell (SSH) Protocol Architecture, 2006. Available online: https://tools.ietf.org/pdf/rfc4251.pdf. Last accessed: 2020-11-06.

[61] Tatu Ylonen and Chris Lonvick. The Secure Shell (SSH) Transport Layer Protocol, 2006. Available online: https://tools.ietf.org/pdf/rfc4253.pdf. Last accessed: 2020-11-06.

Appendix A

The DieHarder testing suite

Table A.1: The Dieharder tests [12]

Test name	Description
Birthdays Test	If the random number generator used to produce a given number is truly random, the test generates a histogram that follows a Poisson distribution by accumulating the results from at least one hundred executions. These results come from counting the number of matching intervals obtained from 512 "birthdays" (by default) drawn on a 24-bit "year" (by default); birthdays and years represent numerical values in this context.
Overlapping 5-Permutations Test	It considers sequences of one million 32-bit numbers and assigns a state to groups conformed by five numbers. The test counts each state's occurrences and calculates the probability that they come from the specified asymptotically normal distribution.
32x32 Binary Rank Test	It randomly creates a 32x32 matrix where each row is a 32-bit number and ranks them with values between zero and thirty-two. The test groups together the ranks equal to or lower than 29; the rest of the ranks remain as they are. This process is repeated 40,000 times and calculates the chi-square on counts for ranks each rank group. This test looks to understand if the matrices values are approximately uniform.
6x8 Binary Rank Test	The test extracts six random 32-bit integers from the tested number, a specified byte is chosen from each. The selected six bytes form a 6x8 binary matrix whose rank (a number between 0 and 6) is determined. The ranks are determined for 100,000 random matrices. The chi-square test is performed on the frequencies of the ranks.
Bitstream Test	It uses a 20-bit-sliding-window to calculate the number of missing 20-bit substrings in a 2^{21} overlapping 20-bit string. If the string is genuinely random, the number of missing words follows an almost exact normal distribution. By default, Dieharder executes this test twenty times.
Diehard Overlapping Pairs Sparse Occupance (OPSO) Test	The OPSO test considers 2-letter words from an alphabet of 1024 letters. Each letter is determined by a specified 10 bits from a 32-bit integer in the number to be tested. OPSO generates 2-letter words and counts the number of missing words, i.e., words which do not appear in the entire sequence. That count is expected to resemble normal distribution.
Diehard Overlapping Quadruples Sparse Occupance (OQSO) Test	Similar to OPSO test, except that it considers 4-letter words from an alphabet of 32 letters, each letter determined by a designated string of 5 consecutive bits.
Diehard DNA Test	The DNA test considers an alphabet of 4 letters:: C,G,A,T, determined by two designated bits in the sequence of random integers being tested. It considers 10-letter words, and similar to OPSO and OQSO test determines distribution of missing words.

Diehard Count the 1s (stream) Test	Each tested number is represented as a stream of bytes. Depending on the frequency of 1's in each byte, a corresponding letter is assigned (only letters A,B, C, D, E are allowed). Each bytes is then converted into a string of 5-letter words consisting of these letters.
Diehard Count the 1s Test (byte)	This test considers files as streams of bytes, each having between zero and eight 1s. The streams provide strings with overlapping 5-letter words categorized from A to E, assigned according to each word's number of 1s. Then, from 4 and 5-letter cell counts, the test calculates the chi-square test and the naive Pearson sums' difference.
Diehard Parking Lot Test	This test considers a square of size 100 to park cars randomly. Suppose there is a crash, the test parks the car in a different location. At each trial, the test lists successful and failed attempts and the number of already parked cars. Instead of graphs, the test uses a value k equal to 3523 with sigma 21.9 to represent the successfully parked cars after 12000 attempts; thus, The value of $\frac{(k-3523)}{21.9}$ represents a standard normal variable which is then input to a Kolmogorov–Smirnov test when transformed into a uniform variable.
Diehard Minimum Distance (2d Spheres)) Test	This test chooses 8000 random points from a square of size 10000. Then, it finds the minimum distance d between the $\frac{(n^2-n)}{2}$) pairs of such points. If they are independent uniform, there is an exponential distribution for d2 with a mean of 0.995. The results from calculating $1 - exp(\frac{-d^2}{0.995})$ should be uniform on [0,1). The test repeats these calculations one hundred times and uses the Kolmogorov–Smirnov (KST) test to check the selected points' uniformity.
Diehard 3d Sphere (Minimum Distance) Test	This test chooses 4000 points randomly from a cube of edge 1000. Then, it centers a sphere at each point's center that reaches the closest point. The volume of the smallest sphere has an exponential distribution with a mean of $\frac{120\pi}{3}$. The test repeats these steps twenty times, and the KST checks those results.
Diehard Squeeze Test	This test takes random integers and converts them to fit uniforms on [0,1). It then goes from $k = 2^{31}$ to one by calculating $k = ceiling(k * U)$; the U value represents the transformed integers. It finds the number of iterations required to get k down to one. The test executes this calculation 100000 times and counts the number of $iterations \leq 6, 7, ..., 47, \geq 48$ and use them to provide a chi-square test for cell frequencies.
Diehard Sums Test	This tests floats a sequence of integers of uniform [0,1) variables. Then it overlaps their sums; these are virually normal with a certain covariance matrix. It then transforms the sums into uniform variables for KST; from these results, the test takes ten p-value and perform a second KST.
Diehard Runs Test	This test takes a sequence of uniform [0,1) variables from floated 32-bit integers, and it counts the runs up and runs down (i.e., comparison between values). The test performs chi-square tests for quadratic forms in the weak inverses of the runs' covariance matrices. These calculations are executed ten times and then repeated. Dieharder uses sequences of length = 100000 and runs a final KST on the generated 100-p values.

Diehard Craps Test	This test plays 200,000 games of craps, counts how many are wins and the number of throws needed to end the games. The wins should be close to a normal distribution with a mean of 200000p and variance $200000p(1-p)$, with $p = \frac{244}{495}$. The number of throws can vary between one and infinity, but the test groups $throws > 21$. Finally, the test calculates the chi-square on cell counts for the number of throws.
Marsaglia and Tsang GCD Test	The test generates 10^7 samples of unsigned integers. From these, it generates their greatest common divisor (GCD) and the number of steps of Euclid's Method k required to find it. Then, it produces a table for the frequency of k between zero and 41, and the occurrences of each GCD. By using the gfsr4, mt19937_1999, rndlxs2 and taus2 RNGs, the test constructs a table of high precision probabilities for the values of k; these four RNGs run the test with each other's tables. The chi-square tests of these distributions generate two p-values per test, and 100 p-values of each are accumulated and subjected to a final KST.
STS Monobit Test	This test is highly effective in finding weak RNGs. It counts the number of bits equal to 1 within random unsigned integers. Then, it compares the count with a precalculated value. Finally, it calculates a p-value using *Perfc()*.
STS Runs Test	Counts the one and zero runs accors samples of bits; the former 01 + 10 and the later 10 + 01. Basically, it counts 01 + 10-bit pairs.
STS Serial Test (Generalized)	This test accumulates the frequency of overlapping n-tuples of bits from random integers and checks if the RNG generates bit-patterns correctly distributed. The expected distribution of n-bit patterns is multinomial with $p = 2^{-n}$.
RGB Bit Distribution Test	Accumulates the frequencies of all n-tuples of bits in a list of random integers and compares the distribution thus generated with the theoretical (binomial) histogram.
RGB Generalized Minimum Distance Test	This test uses n points to determine the test's granularity on an x number of trials at a dimension d; these parameters are selectable to allow users to thoroughly test RNGs while leaving room for higher-order corrections when n and p are large.
RGB Permutations Test	This test counts the order of non-overlapping permutations from random numbers when taken n at a time; where n! is the number of equally possible permutations. Since these samples are independent of each other, the test only calculates one chi-square on the count vector with n! - 1 degree of freedom.
RGB Lagged Sum Test	This test adds all uniform deviates samples drawn from the RNGs and skips lag samples between each run. The samples' mean should be 0.5 * samples with an STD of sqrt(samples / 12). The rest converts all the calculated values into a p-value by using erf() and uses them to runs a final KST.
RGB Kolmogorov-Smirnov Test	This test produces a vector of uniformly deviated samples from the selected RNG. It then applies the Anderson-Darling (AD) or the Kuiper KST (KKST) to evaluate the samples' uniformity. The test runs p-values times, and the KST generates a final p-value. This test does not check the RNG; its purpose is testing the ADKS and KKST tests.

DAB Byte Distribution	This test extracts n independent bytes from each k consecutive words. When the RNG's word is tiny, the test uses overlapped bytes. It increments the indexed counters in each table and uses the lowest and highest bytes and $n - 2$ bytes extracted from the middle. Then, it computes a basic Pearson's chi-square fitting test for the p-value.
DAB Discrete Cosine Transform (DCT) Test	This test uses a DCT on the RNG's output. If the samples are large, the test records the maximum absolute value in each transform, and then it subjects them to a chi-square test for uniformity or independence.
DAB Fill Tree Test	This test uses small binary trees of fixed depth; it fills them up with words drawn from the RNGs. Every time the test cannot place a word within a tree, it records the number of words and their positions. The test rotates the numbers' positions to detect numerical ranges bias such as taking numbers from the high, middle or low bytes. It finally returns p-values calculated from two Pearson chi-squares. The first one calculated determined against the expected values and the second against a uniform distribution at the positions where inserts fail.
DAB Fill Tree 2 Test	Similar to the previous test but at bit-level and uses visited markers instead of insertions. In this test, each bit represents a step within a tree, where zeroes represent steps to the right, and ones represent steps to the left. When taking a step, the test checks if the node is marked; if it is empty, the test marks it as visited. The test continues similarly until the end of the bitstream.
DAB Monobit 2 Test	This test is similar to the Monobit test; it is useful when we do not know what block size to use when checking the RNGs. It tries all sizes of 2^k words with k values between 0 and n. The generator's word size and the sample size in use provide the value for n.