Origin Attribution of RSA Public Keys

Enrico Branca, Farzaneh Abazari, Ronald Rivera Carranza, and Natalia Stakhanova(✉)

Department of Computer Science, University of Saskatchewan, SK, Canada
{enb733,faa851,rer655,natalia}@usask.ca

Abstract. In spite of strong mathematical foundations of cryptographic algorithms, the practical implementations of cryptographic protocols continue to fail. Insufficient entropy, faulty library implementation, API misuse do not only jeopardize the security of cryptographic keys, but also lead to distinct patterns that can result in keys' origin attribution. In this work, we examined attribution of cryptographic keys based on their moduli. We analyzed over 6.5 million keys generated by 43 cryptographic libraries versions on 20 Linux OS versions released over the past 8 years. We showed that with only a few moduli characteristics, we can accurately (with 75% accuracy) attribute an individual key to the originating library. Depending on the library, our approach is sensitive enough to pinpoint the corresponding major, minor, and build release of several libraries that generated an individual key with an accuracy of 81%–98%. We further explore attribution of SSH keys collected from publicly facing IPv4 addresses showing that our approach is able to differentiate individual libraries of RSA keys with 95% accuracy.

Keywords: RSA security · Cryptographic libraries · Attribution

1 Introduction

Secure communication on the Internet is becoming a norm. Nowadays, nearly 90% of all Internet communication is encrypted. While in theory, cryptographic solutions are provably secure, in practice, the security of communication depends on the correctness of implementation of the existing tools that support encryption standards. Over the past decade, numerous studies pointed out weaknesses of cryptographic security of various protocols (TLS/SSL [18], SSH [16], HTTPS [2,10,17]). The majority of these studies investigated insufficient security of generated keys as a main root cause of the problem. Some studies traced the problem to weak random key generators and the lack of entropy [8,13,18], while others noted the improper implementation of cryptographic libraries [11,26,29,37], and pure misuse of cryptographic algorithms, e.g., keys embedded in binary files [12].

As a consequence, the number of studies developed techniques to identify insecure and vulnerable keys. The vast majority of these approaches focus on analysis of binaries that contain vulnerable keys [26,27,33,35], or crypto libraries and APIs that produce those keys [31,37].

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2021
Published by Springer Nature Switzerland AG 2021. All Rights Reserved
J. Garcia-Alfaro et al. (Eds.): SecureComm 2021, LNICST 398, pp. 374–396, 2021.
https://doi.org/10.1007/978-3-030-90019-9_19

Misconfigurations of cryptographic algorithms and cryptographic operations can potentially lead to distinct patterns in the generated keys and can be leveraged in identifying their origins. This observation was first explored in the study by Svenda et al. [37] that noted that combination of implementation decision made in software libraries in a presence of certain hardware is sufficient to identify a probable origin of a key. Their work was further improved by Nemec et al. [31]. Both approaches in their attribution analysis relied on a set of rules defined through a preliminary analysis of biases of the known libraries. In spite of higher accuracy of attribution obtained by [31], none of these approaches were able to attribute an individual key to a specific library, focusing on attribution to groups of similar libraries.

The question that remains is whether *it is feasible to identify an exact origin of an individual cryptographic key.* Addressing this question has direct practical implications. In cryptographic theory, the attribution of keys should not be possible, an accurate tracing of a key to its specific library version allows for fine-grained fingerprinting of cryptographic libraries, which has a number of practical uses from undermining anonymity of Internet users by allowing more accurate profiling of their activities to direct attacks on libraries and protocols [25]. The attribution of keys also implies that these identifiable library implementations embed predictable patterns in the generated keys, thus reducing key space, and allowing for faster key factorization [14].

In this work, we propose a source attribution approach based on the characteristics of RSA key modulus. We analyze the characteristics of RSA public key modulus to understand how much information one needs to trace an individual key to its originating library. The underlying assumption of the source attribution of cryptographic keys is the presence of distinct bit patterns in keys that allows to predict where this key was generated. To quantify these patterns, we derive spatial characteristics of each key moduli to estimate its position in the numerical spectrum and the likelihood that such key may have been generated by a particular library.

To validate our approach, we tested over 6.5 million keys generated by 43 cryptographic libraries versions on 20 Linux OS versions released over the past 8 years. Our experiments show that we can accurately attribute an individual key to the originating library with 75% accuracy with only a few modulus characteristics regardless of its patch level, and its release date. We are further able to produce a fine-grained attribution of a key to the corresponding major, minor, build and in some cases patch releases for several libraries achieving accuracy in the range of 81%–98%.

Our findings suggests that code changes applied to some library implementation between versions leave significant traces in the generated keys, consequently, allowing for accurate origin attribution.

We compare our approach to the most recent study by Nemec et al. [31]. 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 origin 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 homogeneous pool of libraries. We are able to differentiate individual libraries of RSA collected keys with 95% accuracy. For individual versions we obtained 68% to 100% accuracy for most libraries that had a sufficient number of keys. More importantly, we have been able to do this without any prior knowledge on the system, hardware platform or the library that generated them. To summarize, we

1. Propose a source attribution approach that can link an individual cryptographic key to the originating library and its specific version. Our approach does not rely on previous knowledge of cryptographic library's weaknesses.
2. We evaluate and select the top distinctive moduli characteristics that contribute the most in discriminating individual keys.
3. We test the performance of our attribution on a set of keys collected from publicly facing IPv4 addresses. Both sets of generated keys and collected keys are available at https://cyberlab.usask.ca/attributionRSAkeys.html.

This paper is organized as follows: Sect. 2 gives an overview of related work in the field. We briefly introduce the RSA cryptosystem in Sect. 3 and explain our proposed approach in Sect. 4. In Sect. 5, we demonstrate the results of key attribution. Section 6 applies our attribution approach to a real-world keys collected on the Internet.

2 Related Work

In spite of strong mathematical foundations of cryptographic algorithms, the practical implementations of cryptographic protocols continue to fail. The study of real-life spread of vulnerable keys across the Internet by Heninger et al. [18] showed that out of 6.2 million SSH keys collected in the wild 0.03% can be factored within 2 h. Similar results were obtained by Lenstra et al. [24] on the analysis of TLS certificates.

These cryptographic failures rarely occur due to shortcomings of an algorithm's theoretic design or technological advances. Predominantly the compromise of cryptographic protocols happens due to errors in their implementation or due to human oversight in proper configuration or selection of parameters. Lazar et al. [23] examined 269 cryptographic vulnerabilities showing that 83% of bugs were related to misuses of cryptographic libraries, while 17% were bugs in libraries' implementation. Several studies showed that application programming interfaces (APIs) themselves, their complexity and improper default parameters, contribute to the cryptographic misuse [1,30].

A significant number of vulnerabilities in cryptographic keys are stemming from problems of random numbers generators (RNGs). The 2012 study by Heninger et al. [18] attributed the majority of factored RSA keys to memory

constrained devices (such as routers, smart cards, firewalls) that have limited sources for generating appropriate randomness. Indeed, RSA prime factorization arithmetic can be computationally expensive for resource constrained devices. This often leads to various practises that shortcut appropriate key generation and consequently leads to weak public keys that allow an attacker to calculate the private key from a public key.

Yilek et al. [39] examined the spread of keys affected by the highly publicized bug discovered in 2008 in OpenSSL library that generated predictable random numbers. They noted that even after six months of disclosure, weak keys resulting from the buggy implementation were still being issued and widely used. Slow response was also noted by Hastings et al. [17] in their follow-up study that factored 313,000 RSA keys.

Several studies developed techniques to identify insecure and vulnerable keys. The vast majority of these approaches focus on analysis of binaries that contain vulnerable keys [26,27,33,35]. Insufficient entropy, faulty library implementation, API misuse do not only jeopardize the security of cryptographic keys, but also lead to distinct patterns that can result to keys' origin attribution.

Svenda et al. [37] tested if cryptographic hardware cards and libraries comply with the quality and security expectations regarding randomness and resistance against well-known RSA attacks. Within this study they identified seven implementation decisions tied to specific hardware cards and libraries. Cumulatively, these implementation patterns allow to attribute the origin of RSA keys based on the moduli. Their analysis of 60 million generated RSA keys showed the viability of the approach. They were able to correctly label the origin of 40% due to software and hardware differences in their design, implementation choices and faulty RNGs.

Nemec et al. [31] applied the approach developed by Svenda et al. [37] to examine popularity of cryptographic libraries on the Internet. They identified that efficiency improvements, implementation choices and bugs are the main sources of biases when selecting primes p and q, and since these biases can sometimes be observable from the moduli, it is possible to group libraries based on their similarities. The authors showed that it possible to attribute individual keys (with over 94% accuracy) to groups of similar libraries.

Both approaches (i.e., [37] and [31]) in their classification analysis relied on a set of rules defined through preliminary analysis of key misuses of known libraries. Among other things, both approaches required an estimation of prior probabilities for domain where the key to be attributed is coming from. The studies showed that the better this estimate, the better the accuracy of their approaches. Yet, in spite of high accuracy of attribution, none of these approaches were able to attribute keys to individual libraries within groups.

Muslukhov et al. [29] also noted that the improper implementation of cryptographic libraries can leak the origin of a key. Their approach, BinSight, performed a static analysis of Android applications detecting calls to crypto APIs. Similarly to [37] and [31], it was identifying misuses against a set of crypto rules. In this

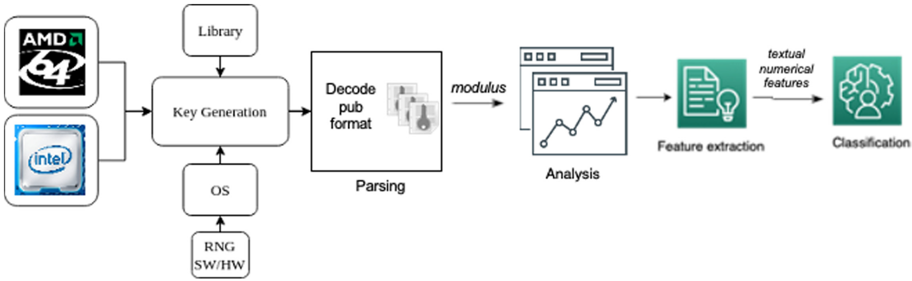


Fig. 1. The flow of the proposed approach

research, we attribute individual keys to the corresponding libraries without a prior knowledge of specific library weaknesses.

3 Background

In this work, we focus on the RSA (Rivest–Shamir–Adleman) algorithm as this is arguably the most popular cryptographic system utilized on the Internet today. RSA is an asymmetric cryptographic algorithm that leverages the fact that while multiplication of large prime numbers is not computationally intensive, factorization of large prime numbers is significantly more complex.

As such an RSA public key is theoretically generated based on two large prime numbers p and q used to calculate the modulus n. Specifically, an RSA key can be generated as follows:

1. Pick two primes p and q to calculate the modulus n, $n = p * q$. Both primes should be large (i.e., $size \geq 1024$), random and $p \neq q$.
2. Calculate $\phi(n) = (p-1)(q-1)$,
3. Select exponent e, $\phi(n)-1$ will serve as an upper limit when selecting a value for e which should be large (i.e., $size \geq 1024$) and random. The value for e should be restricted to $e \in 1, 2, ..., \phi(n) - 1$. In addition, the $GCD(e, \phi(n))$ must be equal to 1 so that we know that they are relatively prime.
4. Calculate private key component d: $de \equiv 1(\text{mod}\phi(n))$.

As a result, an RSA public key $Pub_k = (e, n)$ is represented by an exponent e and a modulus n, while an RSA private key $Priv_k$ is usually a pair (d, n).

The RSA algorithm is implemented in a variety of the cryptographic libraries. For our analysis, we use the most common open-source libraries: OpenSSH, OpenSSL, GnuTLS, and GPG.

4 Analysis Methodology

Several previous studies [31,37] showed that various implementation decisions and shortcuts of cryptographic libraries propagate to RSA keys creating a bias.

In this work, we analyze the characteristics of RSA public key to understand how much information one needs to trace an individual key to its originating library.

The underlying assumption of the source attribution of cryptographic keys is the presence of distinct bit patterns in a single key that allows to predict where this key was generated. Source attribution of the keys does not consider factorizing the modulus n and relies on the numerical and textual features of the modulus, such as a longest repeated substring and a percentage of bits equal to one.

The overview of the attribution approach is illustrated in Fig. 1. The generated cryptographic keys are decoded and parsed to derive a key modulus. As the first step in our analysis, we examine the characteristics of keys' moduli generated from known libraries, estimate the randomness of moduli, establish and analyze the distinctive bit patterns. We use the retrieved patterns in classification analysis to attribute keys to their origin library.

We further analyze contribution of individual patterns to the accuracy of attribution. In our attribution analysis, we employ six classifiers with different logic to understand their effectiveness in discriminating keys originated from different libraries.

4.1 RSA Keys Generation and Parsing

To establish 'ground truth', we generated 6.5 million RSA keys using 4 cryptographic libraries on 9 most widely used Linux distributions released over the past 8 years. We made the design choice to use the cryptographic library version that was shipped with the OS, assuming that this version would have been officially tested and certified by each Linux distribution security teams. Overall, we tested 43 cryptographic libraries versions on 20 versions of Linux distributions. To ensure sufficient entropy for key generation, we enabled software-level random number generator Haveged [36] and used two hardware random number generators: TrueRNG [38] and NeuG [40].

The details of the generated keys are given in Table 2. All keys were generated 2048 bit long as this is the minimum key size recommended by NIST to be used in an RSA algorithm. We verified that all generated keys are valid and NIST standard-compliant, i.e., all moduli are unique, exactly 2048 bits long, and have prime components. To test for primality, we used the Miller-Rabin primality test. To understand the potential randomness of generated keys, we performed the following NIST statistical tests [5]:

Discrete Fourier Transform (DFT) test that detects periodic features (i.e., repetitive patterns that are near each other) in the tested sequence that would indicate a deviation from the assumption of randomness.

Monibit test that checks the proportion of zeroes and ones for the sequence. The purpose of this test is to determine whether the number of ones and zeros in a sequence are approximately the same as would be expected for a truly random sequence.

While a number of studies acknowledged weaknesses of some NIST statistical tests (e.g., Marsaglia's Diehard, and TestU01 [20]), they still serve as one of generic instruments in assessing key randomness.

The results of these tests are given in Table 1. The overwhelming majority of the keys passed both tests. In other words, the produced bit sequences are considered random.

Table 1. The results of NIST statistical tests on the set of generated keys

Test	Number of keys that passed the test	Number of keys that failed the test
Monobit test	6,700,705 (99%)	66,373 (1%)
DFT test	6,674,844 (98%)	92,234 (2%)

4.2 Key Analysis and Representation

Previous study by Nemec et al. [31] identified several rules that cumulatively form a fingerprint allowing to identify a key source (i.e., a group of similar libraries). The rules are specific to the analyzed libraries and are derived from the identified in advance bias of cryptographic libraries. We take a different approach and extract features that are independent of knowledge of the originating library, underlying platform, and operation system.

The intuition of our approach is simple. A bounded numerical space determines a pool of available keys for each library. If any specific rule exists in the key generation process, the generated keys will contain a pattern that reflects the reduced numerical space.

For each key, we consider numerical and textual representation of a modulus and derive features that quantify its randomness and its spatial characteristics.

Numerical Representation. The following types of features are derived:

- *Modulus characteristics*: size, primality of components, entropy.
- *The cutoff value characteristics*: most algorithms for generating RSA modulus set bits in certain positions (usually first two bits are set to 1, and last is set to 1)[1] which defines/reduces a numerical space for selecting potential modulus. For example, for an explanation of how OpenSSL library sets the bits refer to [32]. The *cutoff value* represents the minimum value that can be ever generated for a modulus in such space. We calculate a cutoff value and check its position against an actual integer value of the generated modulus (we refer to it as offset position).
- *Bin position*: helps us determine the preference of the library in choosing a modulus within a specific range. 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

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

Table 2. The summary of the generated keys

OS	OS Version	Year	OpenSSH library	GnuTLS library	GPG library	OpenSSL library
Ubuntu	20.04	2020	8.2p1	3.6.13	2.2.19	1.1.1d
Ubuntu	18.04	2018	7.6p1	3.5.18	2.2.4	1.1.1
Ubuntu	16.04	2016	7.2p2	3.4.10	2.1.11	1.0.2g
Ubuntu	14.04	2014	6.6	*	2.0.22	1.0.1f
Ubuntu	12.04	2012	5.9	*	*	*
Mint	20	2020	8.2p1	3.6.13	2.2.19	1.1.1f
Mint	19	2020	7.6p1	3.5.18	2.2.4	1.1.1h
Fedora	30	2019	8.0p1	3.6.10	2.2.13	1.1.1b
Fedora	23	2015	7.1p1	3.4.5	2.1.7	1.0.2d
Fedora	20	2014	6.3	3.1.16	*	1.0.1e
Fedora	17	2012	5.9	*	*	1.0.0i
Fedora	14	2010	5.5	*	*	1.0.0a
CentOS	8.2.2004	2019	8.0p1	3.6.8	*	1.1.1c
Manjaro	20	2020	8.3p1	3.6.15	2.2.23	1.1.1g
Swift	4.19.0	2018	7.9p1	*	*	1.1.1d
Endeavour	5.8	2020	8.3p1	3.6.15	2.2.23	1.1.1g
Kali	2020.3	2020	8.3p1	3.6.15	2.2.20	1.1.1g
Oracle	R8	2019	8.0p1	3.6.8	*	1.1.1c
Oracle	R7	2017	7.4p1	*	*	1.0.2k
Oracle	R6	2013	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 or their dependencies are no longer available for specific Linux distributions.

value to each key depending on where it falls with respect to its relative section. An analysis of the spatial characteristics of a key moduli (Bin value together with cutoff and its position) can 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 *Brotli* [21] and *Lempel–Ziv–Markov chain (LZMA) compression* [34] algorithms. The 'degree of randomness' can be expressed as 'ratio of compressed file to uncompressed file', where the compression ratio provides a quick way to visualize the randomness used by a library when generating a key.

Textual Representation. To identify and derive all possible bit patterns, we convert the binary representation of a key into a textual format. To exhaustively search for all possible patterns, we employ an overlapping sliding window technique with windows of size $n = 8$ to 256. We derive the following types of textual features for all window sizes:

- *Longest repeated substring (LRS) characteristics*: we estimate the presence and the corresponding characteristics of longest repeated substring patterns within a modulus.
- *0's & 1's characteristics*: we determine and profile a maximum length of continuous zeroes or ones within the string representation of a modulus.
- *Characteristics of mirror patterns.* The modulus' binary string is represented as two halves of equal size, we then perform a comparative analysis of the n bit patterns found in the first half vs the n bit patterns in the second half. This produces features such as *Most significant bits (MSB) - Least significant bit (LSB) pattern* of length n, *mirror pattern*, i.e., n bits of the first half of the modulus found on the second half of the modulus, position and frequency of mirror patterns, etc.

4.3 Classification

We explored performance of 6 classification algorithms: Gaussian Naïve Bayes, Neural Networks, Decision Trees, Discriminant Analysis, Random Forest, and Logistic Regression analysis.

Gaussian Naïve Bayes is based on Bayes' theorem that assumes an independence between features [6]. Even though feature independence assumption rarely holds true, NB models perform surprisingly well in practice [3]. The Gaussian Naïve Bayes classifier is one of its versions that follows a Gaussian distribution and assumes the presence of data with continuous values which is the case in our datasets.

Neural Networks (NN) [28] are a series of algorithms that mimic the operations of a human brain to detect relationships between high volumes of data. Since neural networks can have many layers and parameters with non-linearities, they are very effective at modelling highly complex non-linear relationships. Neural networks operate well with large amounts of training data.

Decision Trees (DT) [7] produces a sequence of rules that can be used to classify the data when a data of features together with its target are given. The decision tree classifier can be unstable because small variations in the data might result in a completely different tree being generated.

Discriminant Analysis model is composed of discriminant functions based on linear combinations of the features that provide the best discrimination between the classes [22]. This model assumes that different classes generate data based on different Gaussian distributions. Linear Discriminant Analysis (LDA) provides multi-class classification which is suitable for our analysis.

Random forest (RF) [19] classifier is an ensemble that fits a number of decision trees on various sub-samples of datasets and uses the average to improve the predictive accuracy of the model.

Logistic Regression(LR) [9] is a linear classifier that predicts probabilities rather than classes. We use multinomial logistic regression classification to calculate the probability of key modulus x belonging to a target class.

Our approach was implemented using the Python language (v 3.8.5) with the scikit-learn library (v 0.23.2). A summary of the classification algorithms'

parameters is given in Table 3. 5-fold cross-validation was employed to measure the accuracy of the machine learning models.

Table 3. Machine learning model parameters

Name	Parameter	Kernel
Gaussian Naive 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 Attribution Results

The results of attributing 6,767,078 keys to the individual libraries are shown in Fig. 2(a). The best average classification accuracy (75% accuracy) is obtained with Random Forest, Logistic Regression, and Neural Network algorithms. The average accuracy is computed based on the accuracy for each class and the number of keys in that class. 5-fold cross-validation was employed to measure the accuracy of the models with 75% of data for training purposes and 25% for testing.

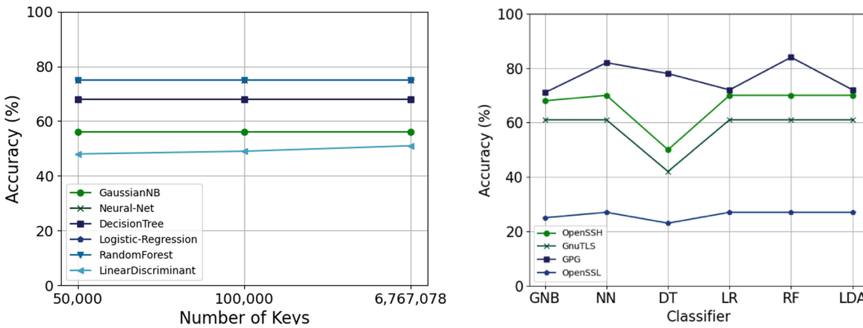
Features. As we anticipated, not all the characteristics extracted from the key material equally contribute to the classification accuracy. To ensure that we keep only features with a measurable impact on the overall accuracy, we have retained features which have an 'Information Gain' (IG) of at least 0.005.

Compared to the overall set of features, we have been able to maintain the same accuracy by selecting the 14 most contributing features (see Appendix). The further analysis of the most significant features shows that while different features are ranked differently for different libraries, the core features remain the same. The core features across various libraries are Brotli compression, LRS pattern, LRS position, Zeroes position, Ones position, Mirror all positions, Mirror position, Mirror all patterns and Bin. Note that only features with windows size $n = 8$ proved to be most significant across all libraries.

One of the challenges that previous studies faced is the necessity to define crypto rules, i.e., weak implementation decisions that effectively lead to bias in the produced keys. The selected characteristics mentioned before are generic and independent from the underlying library.

Attribution Across Library Versions. While the performance of classifiers remain somewhat stable for different amount of keys, we found that the accuracy varies depending on the individual characteristics of the source library.

To further understand the granularity of our approach to discriminate keys, we have evaluated attribution across library versions. Figure 2(b) shows the average accuracy of attributing the individual keys to the libraries' versions. The best average accuracy of 85% is obtained with Random Forest (RF) classification algorithm. Since both experiments showed that RF performs the best in our setting, we further use this classifier in the analysis.



(a) Attribution of keys to their source libraries

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

Fig. 2. The accuracy of attributing generated keys to the originating library and library versions

Table 4 presents more granular results of attributing keys by major and minor release versions with its corresponding libraries[2]. Since we do not have enough key material from GnuTLS, GPG, and OpenSSL libraries to discriminate between major versions, their corresponding experiments were performed for major and minor releases.

We can observe that certain libraries have clearly distinguishable patterns (e.g., GPG, OpenSSH 8.x).

Theoretically, code changes applied to a library that occurred within a major version should not have any significant impact on the generated keys, while major changes that culminate with the release of 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 version.

[2] We refer to library version using a conventional notation of software versioning where each version is represented by major.minor[.build[.patch]].

Table 4. 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

The most difference among versions is produced by GPG library, where it is possible to attribute the key to a specific version with an accuracy ranging from a minimum of 76% for 2.0.x version to a maximum of 98% for 2.1.x version.

The accuracy for other types of libraries varies depending on the library version. For example, we were only able to attribute keys to 3.6 version of GnuTLS library (61%). Our current assumption is that such variability may stem from the changes in the logic or structure of cryptographic primitives implemented in a library.

Fine-grained Origin Attribution. To further understand this phenomenon, we analyzed the release notes for each library[3]. We aggregated the library versions available in our generated set to reflect the modifications in libraries that involve any changes (i.e., improvements) related to random number generation process. As a side note, the GPG release notes did not provide sufficient level of details on what was changed and when, therefore we were not able to derive further groupings for this library.

The results presented in Table 5 give an insight into the variability in attribution of keys, where only major and minor versions are taken into consideration. For example, 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 caused a structural change in produced keys resulting in distinctive patterns in keys generated before 5.9 release. Hence, we were able to distinguish keys generated with earlier versions with 81% accuracy.

[3] GnuTLS: https://gitlab.com/gnutls/gnutls/blob/master/news.
OpenSSH: https://www.openssh.com/releasenotes.html.
GPG: https://gnupg.org/download/release_notes.html.
OpenSSL: https://www.openssl.org/news/changelog.html.

Table 5. 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 2.

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

In some cases, such as GnuTLS before version 3.5.19, or OpenSSH between version 6.8 and 7.8, our approach was unsuccessful in attributing keys to the corresponding library (0% accuracy), which in fact 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 of leaving to the OS the responsibility to ensure a proper initialization during early boot time, negatively impacts the entropy distillation process that leads to the presence of discernable patterns in the resulting cryptographic key.

Comparative Analysis. To better estimate the accuracy and efficacy of our solution, we have decided to compare the performance of our source attribution approach to the performance of the model developed by Nemec et al. [31]. We have therefore implemented and applied their approach on the set of our generated keys. The results are given in Table 6.

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

Table 6. Classification accuracy of Nemec et al. [31] approach on the generated keys

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

6 Internet Scan of IPv4 Hosts

To explore the origins of RSA keys on the Internet, we have performed an attribution of RSA public keys collected from openly available and publicly reachable Internet servers.

6.1 Collected Data

For our analysis, we contacted 220,837 systems over IPv4, with each system geo-located within the Canadian cyber-space and that would accept connections on ports 22 and/or 2222. We collected the keys during 83 d from August to November 2019, collecting keys using the Secure Shell Host (SSH) protocol, by making a single connection to each host and requesting their public SSH-RSA key. Over this period of time we collected 191,976 SSH keys. Among our collected keys, the majority (191,005) were received through SSH v2.0 protocol (Table 7).

The majority of keys (155,107) were generated using the *OpenSSH* library, which is known to be one of the most widely used SSH libraries on the Internet. Among the collected keys, part of them was generated 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 having been generated with OpenSSH 2.x. Furthermore, we also noticed the existence of 89 keys apparently generated by an OpenSSH library version 12.x that seems to be invalid as the most recent version of OpenSSH at the time of this writing is 8.x.

We then proceeded to extract keys components (i.e., modulus and exponent) from the SSH-RSA public keys and compiled a set of 110,798 unique moduli associated with 7 unique exponents. We then organized the SSH-RSA keys according

to their size in bits (i.e., the moduli size) and identified 24,400 keys with a *legacy* or *deprecated* status. Currently, NIST compliant RSA keys are required to have a lengths greater or equal to 2048-bits [4], and in our set 167,576 keys were found to have at least 2048 bits.

In addition to the cryptographic material from RSA keys, we have recorded key collection date, IP address and TCP port, SSH protocol version and SSH banner. When a banner was present, we have parsed such banner to infer the name of the SSH library that answered our connection request, the SSH version of such library, and the presence or absence of High-Performance SSH/SCP patches. Finally, from the key material we computed SSH-MD5 and SSH-SHA256 of each key to use as fingerprints. We refer to these data points as protocol-related features.

Table 7. General statistics of the retrieved keys

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

6.2 Source Attribution

To analyze the performances of our key attribution approach, we designed a set of experiments to analyze the effect of the key size on the accuracy of different types of classifiers. In order to analyze the importance of dataset size, we randomly selected 7 subsets of 191,976 keys collected from the Internet and created sub-samples with a size of 100, 1000, 5000, 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 makes use of stratified k-folding to reduce bias and increase the likelihood of balanced samples.

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

Figure 3 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 a comparable performance with the exception of Gaussian Naive Bayes. With our approach, we are able to reach a significant level of accuracy (over 90% with RF) across all sets of features for 191,976 keys. The best accuracy (95%) was obtained with all, textual alone, and protocol-related together with textual features with RF classifier (Fig. 3 (a), (c) and (f)).

Table 8. 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%

While it is not surprising to see the effect of protocol-related features on attribution, it should be noted that textual features that only retain internal characteristics of a modulus 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.*

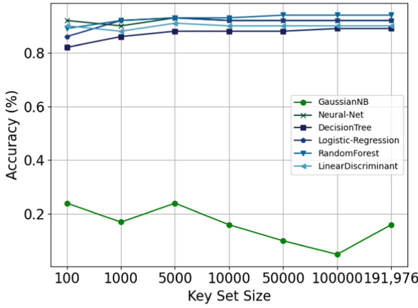
Table 8 clarifies the impact that certain features may have on the accuracy. We are able to reduce the features set of the top 14 features selected in Sect. 5 even further to six characteristics retaining a significant portion of our original accuracy (78% with only 5 modulus characteristics). The description of the top features are provided in Table 8.

If we focus our attention on the size of the data set being used, our analysis reveals that most classifiers perform consistently well across different sample sizes that have at least with 1000 keys (Fig. 3), but seems to perform poorly with sample sizes that have less then 1000 keys, resulting in significant loss of accuracy that is proportional to the size of the sample.

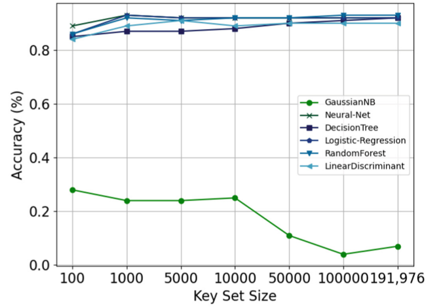
The results of more granular experiments, i.e., attribution to a source library and its version are given in Table 9. In most cases, we are able to attribute keys to their originating library and its version. The performances ranges between 68% to 100% attribution accuracy. The performance is generally lower for libraries/versions with fewer keys which is expected due to lack of sufficient data. For example, in cases of FTP server and Reflection, we were not able to produce any attribution due to very small amount of collected keys (3 and 1, respectively).

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

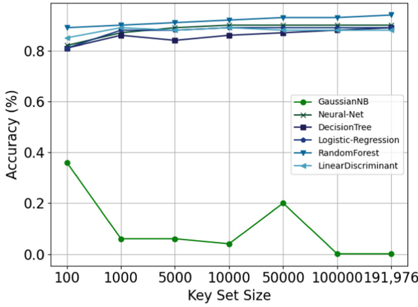
Note that this attribution is based solely on the key's modulus technical characteristics that are independent of library version, patch level, type of kernel used, or hardware platform.



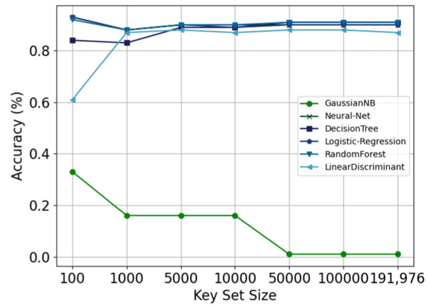
(a) All features



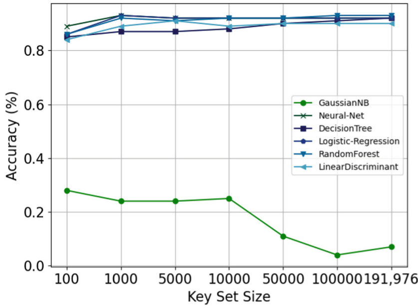
(b) Numerical features



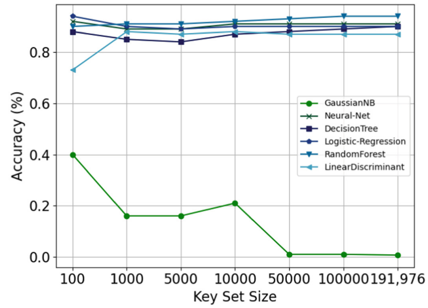
(c) Textual features



(d) Protocol-related features



(e) Protocol-related and numerical features



(f) Protocol-related and textual features

Fig. 3. Attribution accuracy of collected keys by individual library

7 Discussion

Fine-grained attribution of keys to their specific library versions has direct practical implications. Tracing an individual key to an exact version of the cryptographic library that produced it extends our ability to accurately fingerprint cryptographic libraries employed by remote parties, which in turn, undermines security of these systems. Knowledge of a specific library version can

Table 9. Attribution results for the collected SSH keys by the individual library and its version (Random Forest)

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

* As announced in the the SSH banners.

simplify targeted and library-specific attacks. For example, such detailed knowledge would also disclose the technical capabilities of a library in terms of key validation and signing. With this information, an attacker can craft a certificate with characteristics that would result in the acceptance of such certificate as valid instead of it be recognized as forged or malicious (e.g., prefix-collision attacks [25]).

Similarly, an attacker could provide certificate details that would trigger specific bugs in the library parsing routines. Such attack would potentially result in the execution of arbitrary code and in privilege escalation and pivoting attacks, where the cryptographic library is effectively executing code on behalf of a malicious attacker.

The presence of predictable patterns in keys can be also leveraged to expedite the factorization process, effectively undermining the security of this key. Several studies offered heuristics for key factorization in situation when certain bits are set [14]. Another application of this research is finding more information about the type of ransomware [15]

Mitigation of key attribution is challenging at this point. Our results seem to point out that the logic or structure of cryptographic primitives implemented in the libraries are the main cause of the distinct bit patterns in keys. As a result, all keys produced by an identifiable library version will be bound to the same distinct patterns and only modification of library's code can help address them. One potential solution is analysis of quality generated keys. Typically, the output of cryptographic applications is judged by their ability to produce

random or pseudo random sequences. As we showed in this work, these tests do not necessarily recognize the presence of unique patterns, and thus do not fully address the quality of a generated key.

8 Conclusion

Assessing the characteristics of the cryptographic key material in RSA keys through the use of machine learning techniques has allowed us to identify a series of potentially significant aspects of RSA key that are directly linked to the level of security of RSA keys.

The general expectation of cryptographic keys requires the produced key not bear any signs of the originating library. Yet, we have been able to positively associate a percentage of the collected RSA keys not only to a specific library but more worryingly to a specific version of such library.

We generated 6,767,078 keys with different cryptographic libraries, different versions and on different platform. We were able to attribute the keys moduli to the library with an average of 75% accuracy and its version with an average accuracy of 85%. We have been able to find features that can discriminate between keys generated by different libraries, between keys with different major, minor, build and in some case patch versions. For example, we obtained 76% for GPG 2.0.x version and 98% for GPG 2.1.x version.

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

The significance of our work lies in the fact that we can accurately perform attribution of a number of valid, standard-compliant cryptographic keys, both generated and collected from random sources from the Internet. We have evaluated a large number of libraries that span more than a decade, and we found recognizable patterns in every library, regardless of which year the library was published, its patch level, and what is important, its release date. We have been able to perform attribution based solely on the key's modulus technical characteristics that are independent of version, patch level, version of OS, type of kernel used, or class of Linux distribution used for the tests or hardware platform.

9 Appendix

See Table 10.

Table 10. The top 14 features extracted for attribution of generated and collected keys

Feature	Feature type	Description
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
Bin	Numerical	Which bin, between the range of 1 and 100, the value of n falls in
Longest repeated substring (LRS)	String	Longest repeated substring pattern within each string representation of a modulus
LRS position	String	Index positions in which LRS pattern is present within each 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)
Ones position	String	Index positions in which the continuous ones are present within each string representation of a modulus
MSB - LSB pattern	String	The first n bits found in the first half of the modulus string are equal to the last n bits in the second half of the string
Mirror pattern	String	The n bits found in the first half of the modulus string are equal to n bits in the second half of the string (regardless of their position)
Mirror position	String	Index positions of mirror pattern
Mirror all patterns	String	The first MSB n bits found in the remaining string of the modulus (regardless of their position)
Mirror all positions	String	Index positions of mirror all patterns

References

1. Acar, Y., et al.: Comparing the usability of cryptographic apis. In: 2017 IEEE Symposium on Security and Privacy (SP), pp. 154–171 (2017)
2. Acer, M.E., et al.: 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 2017, pp. 1407–1420. ACM, New York (2017)
3. Aly, M.: Survey on multiclass classification methods. Neural Netw. **19**, 1–9 (2005)

4. Barker, E., Chen, L., Roginsky, A., Vassilev, A., Davis, R., Simon, S.: Recommendation for pair-wise key establishment using integer factorization cryptography. Tech. rep., National Institute of Standards and Technology, Gaithersburg (2019). DOI: https://doi.org/10.6028/NIST.SP.800-56Br2,https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Br2.pdf
5. Bassham, L.E., et al.: Sp 800–22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications. Tech. rep., Gaithersburg (2010)
6. Bayes, T.: 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. Philos. Trans. R. Soc. Lond. **53**, 370–418 (1763)
7. Breiman, L., Friedman, J.H., Stone, C.J., Olshen, R.A.: Classification and Regression Trees. Wadsworth International Group, Franklin (1984)
8. Costin, A., Zaddach, J., Francillon, A., Balzarotti, D.: A large-scale analysis of the security of embedded firmwares. In: Proceedings of the 23rd USENIX Conference on Security Symposium, SEC 2014, pp. 95–110. USENIX Association, Berkeley (2014)
9. Cox, D.R., Snell, E.J.: Analysis of Binary Data, vol. 32. CRC Press, Boca Raton (1989)
10. Durumeric, Z., Kasten, J., Bailey, M., Halderman, J.A.: Analysis of the https certificate ecosystem. In: Proceedings of the 2013 Conference on Internet Measurement Conference, IMC 2013, pp. 291–304. ACM, New York (2013)
11. Durumeric, Z., et al.: The matter of heartbleed. In: Proceedings of the 2014 Conference on Internet Measurement Conference, IMC 2014, pp. 475–488. ACM, New York (2014)
12. Egele, M., Brumley, D., Fratantonio, Y., Kruegel, C.: An empirical study of cryptographic misuse in android applications. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS 2013, pp. 73–84. Association for Computing Machinery, New York (2013)
13. Everspaugh, A., Zhai, Y., Jellinek, R., Ristenpart, T., Swift, M.: Not-so-random numbers in virtualized linux and the whirlwind rng. In: 2014 IEEE Symposium on Security and Privacy, pp. 559–574, May 2014
14. Faugère, J.-C., Marinier, R., Renault, G.: Implicit factoring with shared most significant and middle bits. In: Nguyen, P.Q., Pointcheval, D. (eds.) PKC 2010. LNCS, vol. 6056, pp. 70–87. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13013-7_5
15. Fernando, D.W., Komninos, N., Chen, T.: A study on the evolution of ransomware detection using machine learning and deep learning techniques. IoT **1**(2), 551–604 (2020)
16. Gasser, O., Holz, R., Carle, G.: A deeper understanding of SSH: Results from internet-wide scans. In: 2014 IEEE Network Operations and Management Symposium (NOMS), pp. 1–9, May 2014
17. Hastings, M., Fried, J., Heninger, N.: Weak keys remain widespread in network devices. In: Proceedings of the 2016 Internet Measurement Conference, IMC 2016, pp. 49–63. ACM, New York (2016)
18. Heninger, N., Durumeric, Z., Wustrow, E., Halderman, J.A.: Mining your PS and QS: detection of widespread weak keys in network devices. In: Proceedings of 21st USENIX Security Symposium (USENIX Security 12), pp. 205–220. USENIX, Bellevue (2012)
19. Ho, T.K.: Random decision forests. In: Proceedings of 3rd International Conference on Document Analysis and Recognition, vol. 1, pp. 278–282. IEEE (1995)

20. Hurley-Smith, D., Hernandez-Castro, J.: Great expectations: a critique of current approaches to random number generation testing & certification. In: Cremers, C., Lehmann, A. (eds.) Sec. Standardisation Res., pp. 143–163. Springer International Publishing, Cham (2018)
21. IETF: Brotli compressed data format. https://tools.ietf.org/html/rfc7932
22. Lachenbruch, P.A., Goldstein, M.: Discriminant analysis. Biometrics 69–85 (1979)
23. Lazar, D., Chen, H., Wang, X., Zeldovich, N.: Why does cryptographic software fail? a case study and open problems. In: Proceedings of 5th Asia-Pacific Workshop on Systems, APSys 2014. Association for Computing Machinery, New York (2014)
24. Lenstra, A.K., Hughes, J.P., Augier, M., Bos, J.W., Kleinjung, T., Wachter, C.: Ron was wrong, whit is right. IACR Cryptol. ePrint Arch. **2012**, 64 (2012)
25. Leurent, G., Peyrin, T.: Sha-1 is a shambles: First chosen-prefix collision on sha-1 and application to the PGP web of trust. In: 29th USENIX Security Symposium (USENIX Security 20), pp. 1839–1856. USENIX Association, August 2020. https://www.usenix.org/conference/usenixsecurity20/presentation/leurent
26. Li, J., Lin, Z., Caballero, J., Zhang, Y., Gu, D.: K-hunt: pinpointing insecure cryptographic keys from execution traces. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, pp. 412–425. ACM, New York (2018)
27. Li, Y., Zhang, Y., Li, J., Gu, D.: iCryptoTracer: dynamic analysis on misuse of cryptography functions in iOS applications. In: Au, M.H., Carminati, B., Kuo, C.-C.J. (eds.) NSS 2014. LNCS, vol. 8792, pp. 349–362. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11698-3_27
28. McCulloch, W.S., Pitts, W.: A logical calculus of the ideas immanent in nervous activity. Bull. Math. Biophys. **5**(4), 115–133 (1943)
29. Muslukhov, I., Boshmaf, Y., Beznosov, K.: Source attribution of cryptographic api misuse in android applications. In: Proceedings of the 2018 on Asia Conference on Computer and Communications Security, ASIACCS 2018, pp. 133–146. Association for Computing Machinery, New York (2018)
30. Nadi, S., Krüger, S., Mezini, M., Bodden, E.: Jumping through hoops: why do java developers struggle with cryptography apis? In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, pp. 935–946. Association for Computing Machinery, New York (2016)
31. Nemec, M., Klinec, D., Svenda, P., Sekan, P., Matyas, V.: Measuring popularity of cryptographic libraries in internet-wide scans. In: Proceedings of the 33rd Annual Computer Security Applications Conference, ACSAC 2017, pp. 162–175. ACM, New York (2017)
32. OpenSSL: Bn_generate_prime (2021). https://www.openssl.org/docs/man1.1.1/man3/BN_generate_prime.html
33. Piccolboni, L., Di Guglielmo, G., Carloni, L.P., Sethumadhavan, S.: Crylogger: Detecting crypto misuses dynamically. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P). IEEE (2021)
34. Python: The Lempel–Ziv–Markov chain (LZMA) compression algorithm. https://docs.python.org/3/library/lzma.html
35. Rahaman, S., et al.: 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 2019, pp. 2455–2472. Association for Computing Machinery, New York (2019)
36. Seznec, A., Sendrier, N.: Havege: a user-level software heuristic for generating empirically strong random numbers. ACM Trans. Model. Comput. Simul. **13**(4), 334–346 (2003)

37. Svenda, P., et al.: The million-key question—investigating the origins of RSA public keys. In: Proceedings of 25th USENIX Security Symposium (USENIX Security 16), pp. 893–910. USENIX Association, Austin, August 2016
38. ubld.it: TrueRNG. https://hackaday.io/project/630-truerng
39. Yilek, S., Rescorla, E., Shacham, H., Enright, B., Savage, S.: When private keys are public: results from the 2008 debian openssl vulnerability. In: Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement, IMC 2009, pp. 15–27. Association for Computing Machinery, New York (2009)
40. Yutaka, N.: NeuG: a true random number generator implementation. Tech. rep. (2015)