FI MU

Faculty of Informatics
Masaryk University, Brno

The Million-Key Question - Investigating the Origins of RSA Public Keys

by

Petr Švenda
Matúš Nemec
Peter Sekan
Rudolf Kvašňovský
David Formánek
David Komárek
Václav Matyáš

FI MU Report Series **FIMU-RS-2016-03**

 July 2016

Publications in the FI MU Report Series are in general accessible via WWW:

`http://www.fi.muni.cz/reports/`

Further information can be obtained by contacting:

Faculty of Informatics
Masaryk University
Botanická 68a
602 00 Brno
Czech Republic

The Million-Key Question - Investigating the Origins of RSA Public Keys

Petr Švenda, Matúš Nemec, Peter Sekan, Rudolf Kvašňovský,
David Formánek, David Komárek and Vashek Matyáš
Masaryk University, Czech Republic

August 5, 2016

Abstract

Can bits of an RSA public key leak information about design and implementation choices such as the prime generation algorithm? We analysed over 60 million freshly generated key pairs from 22 open- and closed-source libraries and from 16 different smartcards, revealing significant leakage. The bias introduced by different choices is sufficiently large to classify a probable library or smartcard with high accuracy based only on the values of public keys. Such a classification can be used to decrease the anonymity set of users of anonymous mailers or operators of linked Tor hidden services, to quickly detect keys from the same vulnerable library or to verify a claim of use of secure hardware by a remote party. The classification of the key origins of more than 10 million RSA-based IPv4 TLS keys and 1.4 million PGP keys also provides an independent estimation of the libraries that are most commonly used to generate the keys found on the Internet.

Our broad inspection provides a sanity check and deep insight regarding which of the recommendations for RSA key pair generation are followed in practice, including closed-source libraries and smartcards. The inspection was not limited only to public part of a RSA keypair – the properties of private key were inspected including factorization of $p-1$ and $p+1$ for large number of 512-bit RSA keys followed by discussion of relevant factorization attacks [1].

[1]Full details, paper supplementary material, datasets and author contact information can be found at http://crcs.cz/papers/usenix2016. This technical report is an extended version of the paper published at the 25th USENIX Security Symposium 2016 [49].

Contents

1 Introduction

The RSA key pair generation process is a crucial part of RSA algorithm usage, and there are many existing (and sometimes conflicting) recommendations regarding how to select suitable primes p and q [17, 21, 36, 31, 32] to be later used to compute the private key and public modulus. Once these primes have been selected, modulus computation is very simple: $n = p \cdot q$, with the public exponent usually fixed to the value 65 537. But can the modulus n itself leak information about the design and implementation choices previously used to generate the primes p and q? Trivially, the length of the used primes is directly observable. Interestingly, more subtle leakage was also discovered by Mironov [34] for primes generated by the OpenSSL library, which unwantedly avoids small factors of up to 17 863 from $p - 1$ because of a coding omission. Such a property itself is not a security vulnerability (the key space is decreased only negligibly), but it results in sufficiently significant fingerprinting of all generated primes that OpenSSL can be identified as their origin with high confidence. Mironov used this observation to identify the sources of the primes of factorizable keys found by [20]. But can the origins of keys be identified only from the modulus n, even when n cannot be factorized and the values of the corresponding primes are not known?

To answer this question, we generated a large number of RSA key pairs from 22 software libraries (both open-source and closed-source) and 16 different cryptographic smartcards from 6 different manufacturers, exported both the private and public components, and analysed the obtained values in detail. As a result, we identified seven design and implementation decisions that directly fingerprint not only the primes but also the resulting public modulus: 1) Direct manipulation of the primes' highest bits. 2) Use of a specific method to construct strong or provable primes instead of randomly selected or uniformly generated primes. 3) Avoidance of small factors in $p-1$ and $q-1$. 4) Requirement for moduli to be Blum integers. 5) Restriction of the primes' bit length. 6) Type of action after candidate prime rejection. 7) Use of another non-traditional algorithm – functionally unknown, but statistically observable.

As different design and implementation choices are made for different libraries and smartcards (cards) with regard to these criteria, a cumulative fingerprint is sufficient to identify a probable key origin even when only the public key modulus is available. The average classification accuracy on the test set was greater than 73% even for a single classified key modulus when a hit within the top 3 matches was accepted[2]. When more keys from the same (unknown) source were classified together, the analysis of as few as ten keys allowed the correct origin to be identified as the top single match in more than 85% of cases. When five keys from the same source were available and a hit within the top 3 matches was accepted, the classification accuracy was over 97%.

We used the proposed probabilistic classifier to classify RSA keys collected from the IPv4 HTTPS/TLS [15], Certificate Transparency [16] and PGP [54] datasets and identified probable sources for the listed keys. With no prior assumptions regarding the source probability for the keys listed in the IPv4 HTTPS/TLS dataset [15], a remarkably close match to the current market share of web servers with the known underlying software library was achieved – providing an independent verification of the classification success rate of our method.

The optimal and most secure way of generating RSA key pairs is still under discussion. Our wide-scale analysis also provides a sanity check concerning how closely the various recommendations are followed in practice for software libraries and smartcards and what the impact on the resulting prime values is, even when this impact is not observably manifested in the public key value. We identified multiple cases of unnecessarily decreased entropy in the generated keys (although this was not exploitable for practical factorization) and a generic implementation error pattern leading to predictable keys in a small percentage (0.05%) of cases for one type of card.

Surprisingly little has been published regarding how key pairs are generated on cryptographic cards. In the case of open-source libraries such as OpenSSL, one can inspect the source code. However, this option is not available for cards, for which the documentation of the generation algorithm is confidential and neither the source code nor the binary is available for review. To inspect these black-box implementations, we utilized the sidechannels of time and power consumption (in addition to the exported raw key values). When this side-channel information was combined with the available knowledge and observed characteristics of open-source libraries, the approximate key pair generation process could also be established for these black-box implementations.

This technical report is organized as follows:

After a brief summary of the RSA cryptosystem, the rest of Section 1 describes the methodology used in this study. Section 2 provides the overview of relevant attacks

[2]The correct library is listed within the first three most probable groups of distinct sources identified by the classification algorithm, as described in Section 5.1.1.

against the RSA cryptosystem. Review of relevant algorithms for RSA key generation together with inspection of source code of software libraries are provided in Section 3. Section 4 provides a discussion of the observed properties of the generated keys. Section 5 describes the modulus classification method and its results on large real-world key sets, the practical impact and mitigation of the discovered information disclosure vulnerability. Additional analysis performed for black-box implementations on cards and a discussion of the practical impact of a faulty/biased random number generator are presented in Section 6. The inspection of keypair generation process in closed-source smartcard by means of power- and time-consumption side channel is available in Section 7. Finally, conclusions are offered in Section 8.

1.1 RSA cryptosystem

To use the RSA algorithm, one must generate a key:

1. Select two distinct large primes[3] p and q.
2. Compute $n = p \cdot q$ and $\varphi(n) = (p-1)(q-1)$.
3. Choose a public exponent $e < \varphi(n)$ that is coprime to $\varphi(n)$.
4. Compute the private exponent d as $e^{-1} \bmod \varphi(n)$.

The pair (e, n) is the public key; either (d, n) serves as the secret private key, or (p, q) can be used ((d, n) can be calculated from (p, q, e) and vice versa).

The encryption exponent e is typically chosen as a fixed value with low Hamming weight (e.g., 3, 17, 65537) to speed up encryption. In such case, new primes are generated until they are coprime to the exponent.

The *universal private exponent* can be computed as $d = e^{-1} \bmod \lambda(p-1, q-1)$, where λ is the Carmichael function, which is numerically identical to the Euler's function for primes. The function of a composite number $n = p \cdot q$ can be computed as the least common multiple of $\lambda(p)$ and $\lambda(q)$, which is $p-1$ and $q-1$, respectively. Since $\lambda(n)$ is a proper divisor of $\phi(n)$, the private exponent will be smaller. In fact, any exponent $d + k \cdot \lambda(n)$, $k \in \{0, 1, \cdots\}$ will work for RSA decryption.

An alternative representation of the private key is a quintuple (p, q, dP, dQ, $qInv$). It is used when the Chinese remainder theorem (CRT) is applied to RSA. The benefits of CRT for faster decryption were discovered by [42]. The parameters p and q are the

[3] Generated randomly, but possibly with certain required properties, as we will see later.

factors of the modulus, dP and dQ are the CRT exponents of the first and the second factor, respectively. The CRT coefficient is denoted $qInv$. Conversion between the two representations is simple. No additional requirements are placed when generating CRT keys, hence any key can be used with or without CRT.

Some definitions [45, 23] allow for more factors of the modulus, however the multi-prime variant of RSA is rarely used in practice, as we observed from the libraries.

1.2 The RSA cryptographic primitives

The RSA scheme is built on the following cryptographic primitives (mathematical operations):

- The *RSA encryption* takes a message m represented as a positive integer between 0 and $n - 1$. The ciphertext c is $c = m^e \bmod n$.
- The *RSA decryption* converts the ciphertext to the message $m = c^d \bmod n$, assuming that the RSA private key is valid.
- The *RSA signature* takes a message m (typically a message digest) represented as a positive integer between 0 and $n - 1$. The signature s is computed as $s = m^d \bmod n$.
- The *RSA signature verification* converts the signature to the message (message digest) $m = s^e \bmod n$, assuming that the RSA public key is valid.

In practice, the CRT representation of the private key is used to benefit from faster decryption. The primitives, as described, suffer from several attacks (e.g., the Hastad's broadcast attack and the Franklin-Reiter related message attack [8]), therefore padding schemes are necessary for real implementations [45, 23].

1.3 Card usage scenarios

RSA can be used with or without secure hardware (cards) in following principal modes:
I. No secure hardware – The RSA key pair is generated in a software library on a host with the private key stored and used without additional protection of a secure hardware. The process of key pair generation can be verified (if an open-source library is used). Availability of good random generator is crucial. When the host is compromised, the private key is exposed.

II. Generate and export – Both public and private parts are exported to the host. An on-card truly random number generator is utilized to generate a secure key pair with enough entropy. The follow-up usage on the host system after export then allows for fast operations with the private key, but also with the disadvantage of potential host compromise.
III. Import and use – The RSA key pair is generated in a software library with the private key later imported into a card. The private key value is then protected by the card during transport and usage. The private key value is vulnerable during the short period before the private key is transferred to the card and erased from the host.
IV. Generate and use – Generate key pair on-card and then export only the public part. The private key is protected against the host compromise, but due to a closed environment, the card must be trusted to generate secure keys and handle them correctly during their use as only very limited checks are possible on the exported public part.

Our analyses of keys with card origin concern mainly modes II and IV, where the key pair is generated on a card. The analysis of multiple open-source libraries also addresses modes I and III. We focus on the properties of *generate* part, not the *use* part.

1.4 Analysis methodology

Our purpose was to verify whether the RSA key pairs generated from software libraries and on cards provide the desired quality and security with respect to the expectations of randomness and resilience to common attacks. This is not an easy task. Leaving aside the fact that factorization is believed but has not been proven to be an NP-hard problem, one can show that a key is not secure by factorizing the corresponding modulus or by showing that its security margin against a given factorization method is lower than claimed, e.g., if the factors of $p - 1$ prove to be 60 bits or shorter, then the key can be efficiently factorized using Pollard's $p - 1$ method with only modest computation power regardless of the length of the modulus. We attempted to identify the characteristics of the generated keys and to verify whether they possessed exploitable properties that would allow an attacker to recover the private key from a public key or at least decreased the assumed security level of the key. Furthermore, we wished to deduce the reason for the presence of these properties and the process responsible for introducing them. The impact of the techniques used on the properties of the produced public keys was also investigated. We used the following methodology:

1. Establish the characteristics of keys generated from open-source cryptographic libraries with known implementations.
2. Gather a large number of RSA key pairs from cryptographic software libraries and cards (one million from each).
3. Compare the keys originating from open-source libraries and black-box implementations and discuss the causes of any observed similarities and differences (e.g., the distribution of the prime factors of $p - 1$).
4. Analyse the generated keys using multiple statistical techniques (e.g., calculate the distribution of the most significant bytes of the primes).

Throughout this paper, we will use the term *source* (of keys) when referring to both software libraries and cards.

1.5 Analysis of black-box implementations

To obtain representative results of the key generation procedures used in cards (for which we could not inspect the source codes), we investigated 16 different types of cards from 6 different established card manufacturers (2×Gemalto, 6×NXP, 1×Infineon, 3×Giesecke & Devrient (G&D), 2×Feitian and 2×Oberthur) developed using the widely used JavaCard platform. The key pair generation process itself is implemented at a lower level, with JavaCard merely providing an interface for calling relevant methods. For each type of card (e.g., NXP J2D081), three physical cards were tested to detect any potential differences among physical cards of the same type (throughout the entire analysis, no such difference was ever detected). Each card was programmed with an application enabling the generation and export of an RSA key pair (using the `KeyPair.generateKey()` method) and truly random data (using the `RandomData.generate()` method).

We focused primarily on the analysis of RSA keys of three different lengths – 512, 1024 and 2048 bits. Each card was repeatedly asked to generate new RSA 512-bit key pairs until one million key pairs had been generated or the card stopped responding. The time required to create these key pairs was measured, and both the public (the modulus n and the exponent e) and private (the primes p and q and the private exponent d) components were exported from the card for subsequent analyses. No card reset was performed between key pair generations. In the ideal case, three times one million key pairs were extracted for every card type. The same process was repeated for RSA key pairs with 1024-bit moduli but for only 50 000 key pairs, as the key generation process takes progressively longer for longer keys. The patterns observed from the analyses performed on the 512-bit keys was used to verify the key set with longer keys[4].

Surprisingly, we found substantial differences in the intervals from which primes were chosen. In some cases, non-uniform distributions of the primes hinted that the prime generation algorithms are also different to those used in the software libraries. Several methods adopted in software libraries, such as incremental search, seem to be suitable even for limited-resource systems. This argument is supported by a patent application [41] by G&D, one of the manufacturers of the examined cards. All tested cards from this manufacturer produced Blum integers, as described in the patent, and these integers were distributed uniformly, as expected from the incremental search method.

A duration of approximately 2-3 weeks was typically required to generate one million key pairs from a single card, and we used up to 20 card readers gathering keys in parallel. Not all cards were able to generate all required keys or random data, stopping with a non-specific error (`0x6F00`) or becoming permanently non-responsive after a certain period. In total, we gathered more than 30 million card-generated RSA key pairs[5]. Power consumption traces were captured for a small number of instances of the key pair generation process.

In addition, 100 MB streams of truly random data were extracted from each card for tests of statistical randomness. When a problem was detected (i.e., the data failed one or more statistical tests), a 1 GB stream was generated for fine-grained verification tests.

[4]For example, one can quickly verify whether a smaller number of factorized values of $p - 1$ from 1024-bit RSA keys fit the distribution extrapolated from 512-bit keys.

[5]The entire dataset is available for further research at [55].

2 Attacks against RSA cryptosystem

The basic form of attack on the RSA cryptosystem is modulus factorization, which is currently computationally unfeasible or at least extremely difficult if p and q are sufficiently large (512 bits or more) and a general algorithm such as the number field sieve (NFS) or the older quadratic sieve (MPQS) is used. However, special properties of the primes enable more efficient factorization, and measures may be taken in the key pair generation process to attempt to prevent the use of such primes.

Despite the existence of many special-purpose algorithms, the easiest way to factor a modulus created as the product of two randomly generated primes is usually to use the NFS algorithm. Nevertheless, using special primes may potentially thwart such factorization attacks, and some standards, such as ANSI X9.31 [22] and FIPS 186-4 [36], require the use of primes with certain properties (e.g., $p-1$ and $p+1$ must have at least one large factor). Other special algorithms, such as *Pollard's rho* method and the *Lenstra elliptic curve* method, are impractical for factoring a product of two large primes.

Although RSA factorization is considered to be an NP problem if keys that fulfil the above conditions are used, practical attacks, often relying on a faulty random generator, nevertheless exist. Insufficient entropy, primarily in routers and embedded devices, leads to weak and factorizable keys [20]. A faulty card random number generator has produced weak keys for Taiwanese citizens [4], and supposedly secure cryptographic tokens have been known to produce corrupted or significantly biased keys and random streams [10].

Implementation attacks can also compromise private keys based on leakage in side channels of timing [12] or power [26]. Active attacks based on fault induction [50] or exploits aimed at message formatting [2, 7] enable the recovery of private key values. We largely excluded these classes of attacks from the scope of our analysis, focusing only on key generation.

2.1 Attacks on RSA keys in more details

Rivest and Silverman introduced the notion of a *weak key* into the context of the RSA algorithm [44]. *General-purpose attacks* do not rely on the choice of the key and are always successful. Hence the purpose of this section is to show different types of weak keys abused by *special-purpose attacks*, that use a specific property of the key to break it. These attacks are successful only under certain conditions.

Specifically, we will consider factorization of RSA moduli with weak primes, methods that use improperly selected parameters of the keys and methods that were successfully used in large scale attacks. Some factorization methods are not suitable for RSA integers (such as the Pollard's rho algorithm), since they are not able to handle integers with two factors of similar length.

An attacker may first try to mount a special-purpose attacks, hoping that the key is weak. If not successful in a predetermined amount of time, the attacker moves to a more complex general-purpose attack.

2.1.1 Pollard's $p-1$ factorization algorithm

An integer is called *B-smooth* or *smooth with respect to a bound* B, if all its prime factors are $\leq B$.

Pollard's $p-1$ factorization algorithm [38] can efficiently find a factor of a composite number n, if for some prime factor p of n, $p-1$ is B-smooth for a reasonably small B.

For any multiple m of $p-1$ and some a, $\mathrm{GCD}(a,p) = 1$, Fermat's little theorem implies $a^m \equiv 1 \pmod{p}$. Therefore if $f = \mathrm{GCD}(a^m - 1, n)$, then p divides f.

One first guesses the smoothness bound B, then computes the product of all prime powers less than B as $m = \prod_{\text{primes } q \leq B} q^{\log_q n}$. The algorithm returns a factor of n if $p-1$ is indeed B-smooth, otherwise it reports failure.

1. Choose (a random) a, $1 < a < n$.

2. Compute $f = \mathrm{GCD}(a, n)$. If $f > 1$, return f.

3. Compute $x = a^m \pmod{n}$.

4. Compute $f = \mathrm{GCD}(x-1, n)$. If $f > 1$, return f.

5. Could not find a factor of n.

Mitigation. The attack requires B-smooth $p-1$ or $q-1$. If they are both generated to have a large prime factor, the bound B will not be reachable by an attacker.

Probability of success. The probability of success depends on the definition of large factor. The algorithm requires $O\left(\frac{B\ln(n)}{\ln B}\right)$ modular multiplications [33]. Then it is sufficient to estimate what bound B is safe from a motivated attacker. For example, FIPS

Size of p (in bits)	Size of B (in bits)	Probability, that random $p-1$ is B-smooth	
		u^{-u}	Knuth
256	100	$2.56^{-2.56} \approx 9.01 \cdot 10^{-2}$	$1.30 \cdot 10^{-1} \approx 2^{-3}$
512	100	$5.12^{-5.12} \approx 2.34 \cdot 10^{-4}$	$3.55 \cdot 10^{-4} < 2^{-11}$
1024	140	$7.314^{-7.314} \approx 4.78 \cdot 10^{-7}$	$< 8.75 \cdot 10^{-7} < 2^{-20}$
1536	170	$9.035^{-9.035} \approx 2.31 \cdot 10^{-9}$	$< 1.02 \cdot 10^{-9} < 2^{-29}$

Table 1: *Estimate of the probability that a random prime does not pass the requirement, that* $p-1$ *has a large factor. The size of factors is taken from FIPS 186-4 (except for 256-bit primes, since the standard does not allow 512-bit keys). Two approximations for B-smoothness are used,* u^{-u}*, where all prime factors of* n *are* $\leq n^{\frac{1}{u}}$ *and a more precise approximation given by Knuth and Trabb Pardo in [25]. Since an RSA modulus requires two primes and* $p+1$ *must also have large factors, random keys will be FIPS-compliant with quarter the probability.*

186-4 recommends factors of length > 140 bits, for 2048-bit keys. We estimated the probability that a random 1024-bit number fails this requirement to be $< 2^{-20}$.

Probability, that p is a strong prime. Dickman function $\rho(u)$ gives the probability, that for a real $u \geq 1$ all prime factors of n are $\leq n^{\frac{1}{u}}$. An approximation is $\rho(u) \approx u^{-u}$.

The minimum length of large factor can be chosen according to FIPS 186-4 [36, Table B.1]. Table 1 gives the approximate probabilities, that for a random prime p, $p-1$ will be B-smooth for bound given by FIPS 186-4. Similar probabilities apply for B-smoothness of $p+1$. Assuming the sizes of largest factors of $p-1$ and $p+1$ are independent, the probabilities should be doubled. It should be noted that the requirements are strong and much smaller smoothness bounds would suffice to protect against special-purpose factorization in practice.

2.1.2 Williams' $p+1$ factorization algorithm

Williams' $p+1$ factorization algorithm [52] is a special-purpose method similar to the Pollard's $p-1$ method. It factors n if $p+1$ is B-smooth for some prime factor p of n. The computation is based on Lucas sequences.

Mitigation and feasibility. Both $p+1$ and $q+1$ must have at least one large (distinct) factor. The attack succeeds on random keys with probability similar to the $p-1$ attack.

2.1.3 Fermat's factorization method

The Fermat's factorization algorithm belongs to a family of square factoring methods. In general, if $a^2 \equiv b^2 \pmod{n}$ and $a \not\equiv \pm b \pmod{n}$, then n divides the difference of squares $a^2 - b^2 = (a+b) \cdot (a-b)$. Since n does not divide neither $(a+b)$ nor $(a-b)$, $\mathrm{GCD}(a+b, n)$ and $\mathrm{GCD}(a-b, n)$ are factors of n.

Fermat's factorization method tries to find a such that $a^2 - n = b^2$ for some integer b. Then n is factored as $n = (a+b) \cdot (a-b)$.

The algorithm in pseudocode follows:

1. $a = \lceil \sqrt{n} \rceil$.
2. Compute $x = a^2 - n$.
3. While x is not a square, repeat:
   3.1 $x = x + 2a + 1$.
   3.2 $a = a + 1$.
4. Return factors $(a + \sqrt{x})$ and $(a - \sqrt{x})$.

Mitigation. The algorithm is usually published with the initial guess $a = \lceil \sqrt{n} \rceil$, which works for every composite n. However, large primes p and q would have to be very close to $\lceil \sqrt{n} \rceil$ in order to find them in reasonable time. If the extremely unlikely event occurs, new key (or a replacement prime) should be generated. It is possible to search in an interval around any a using this method. Hence the primes should be selected from large enough intervals.

Probability of success. To prevent this attack, p and q should not be too close together. FIPS 186-4 and ANSI X9.44 require the difference $|p - q|$ to be larger than $2^{\frac{k}{2}-100}$, where k is the size of the key. To fail this requirement, two primes of same length would have to be identical in all of their first 100 bits. Assuming uniform distribution of prime generator, the probability is approximately 2^{-100}, neglecting a few (1 or 2) upper bits, that may be fixed.

2.1.4 Lehman's improvement to Fermat's method

Let the ratio of p/q be near a ratio of two small numbers r/s. Then an attacker computes $nrs = ps \cdot qr$, ps and qr are close to $\sqrt{nrs}$ and Fermat's method will efficiently find them. Then $\mathrm{GCD}(ps, n) = p$ and $\mathrm{GCD}(qr, n) = q$.

Lehman discovered a systematic way of choosing the rational numbers [28].

Mitigation and feasibility. After computing p and q, one might check if the ratio p/q is near a ratio of two small integers. In such case a new prime or a new key is generated. This requirement appeared in ANSI X9.31-1997 but the probability of small ratio is negligible [47]. The check is not performed in any analyzed cryptographic library.

2.1.5 General-purpose factorization methods

Before general-purpose factorization methods were known, it was often recommended to actively avoid properties of the primes that would make any of the previous factorization attacks feasible. As we have already shown, the special-purpose factoring attacks succeed with negligible probability for practically large RSA keys. General-purpose attacks do not rely on any conditions.

Elliptic curve factorization. Lenstra's elliptic curve factorization method [30] (ECM) generalizes Pollard's $p - 1$ method. ECM replaces $\mathbb{Z}_p^*$ (which has order $p - 1$) with a random elliptic curve group over $\mathbb{Z}_p$. The algorithm will factor n with high probability, if the order of the randomly selected group is smooth with respect to some small bound. Otherwise the method may be repeated with a different random group. The group orders are uniformly distributed in $\left[p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p}\right]$ [33]. The method is not as efficient as the quadratic sieve.

Quadratic sieve factorization. The quadratic sieve factorization invented by Pomerance [40] searches for a congruence of squares modulo n, but does so systematically, as opposed to random square factoring methods.

While Fermat's method searches for an a such that $x = a^2 - n$ is a square ($x = b^2$), quadratic sieve computes $a_i^2 \bmod n$ for several a_i and then searches for a subset $\{a_1, \ldots, a_j\}$ of a_i whose product $(a_1^2 \bmod n \cdot \cdots \cdot a_j^2 \bmod n)$ is a square. If found, it yields a congruence of squares and thus a factorization of n:

$$(a_1^2 \bmod n \cdot \cdots \cdot a_j^2 \bmod n) \equiv ((a_1 \cdot \cdots \cdot a_j) \bmod n)^2 \pmod{n}$$

Number field sieve factorization. General number field sieve (GNFS or just NFS), originally proposed by Pollard [39], is the most efficient general factorization method for large numbers. As the quadratic sieve method, the algorithm searches for a congruence of squares modulo n. The algorithm is considerably more complicated than QS, but it achieves large speed-up in factorization numbers larger than approximately 350 bits. The current record of factoring 768-bit RSA modulus was achieved with the NFS [24].

2.1.6 Attacks on small private exponent

Wiener [51] devised an attack based on continued fractions which makes it possible to recover the private exponent d from public information, if d is small ($d < n^{\frac{1}{4}}$).

Boneh and Durfee [9] later extended the result to $d < n^{0.292}$ using lattice basis reduction and estimated the correct bound as $d < n^{\frac{1}{2}}$.

Mitigation. The private exponent is small ($d < n^{\frac{1}{2}}$) with very low probability [36]. In such case a new key should be generated.

2.1.7 Cycling attacks

RSA encryption is a permutation on the message space $\{0, 1, \ldots, n-1\}$, hence there always exists some k, such that $c^{e^k} \equiv c \pmod{n}$. Then the message can be recovered by repeated encryption, until the value k is found. The message is computed as $m \equiv c^{e^{k-1}} \pmod{n}$. A generalized version of the cycling attack either factors n or (much less frequently) succeeds as the basic cycling attack [33].

Mitigation and feasibility. Some authors use the cycling attacks to justify the requirement for strong primes [19, 33]. Since this attack typically gives a factorization of the modulus or the private key (from which the prime factors can be determined), it is assumed that it performs no better than other factorization methods. A more precise analysis in [44] shows that the attack is extremely unlikely to succeed.

2.1.8 Small public exponents and Coppersmith's attack

Small exponents are desirable to speed up encryption. The exponent $e = 3$ was commonly used. If the message m is short and the ciphertext is shorter than the modulus ($c = m^3 < n$), the message can be recovered by integer cube root. If three recipients with different moduli and common public exponent $e = 3$ receive the same message, an eavesdropper can compute the message using the CRT (the Hastad's broadcast attack [8]). Protection against these and related attacks rely on salting the messages with different random strings.

Coppersmith [14] published an attack, where the knowledge of two thirds of a message encrypted with public exponent 3 enables an attacker to recover the message. Hence if the padding is less than a third of the message, the attack is still applicable.

Mitigation. Even if a padding scheme is used, do not use short public exponents. A slightly larger exponent with low Hamming weight (e.g., $2^{16} + 1 = 65537$) still provides fast encryption and there are fewer known attacks against RSA with such exponents.

2.1.9 Attacks on keys generated with low entropy

Algorithms for prime number generation and PRNGs are deterministic. When given the same seed, they will generate the same prime. For correctly randomly seeded generator, it is extremely unlikely that two parties will obtain identical primes. However, when a PRNG is incorrectly seeded, the probability of shared factors may become much higher than anticipated (e.g., consider the shared keys generated due to a bug with low entropy seed in OpenSSL in Debian [3]). Other problems arise from malfunctioning PRNGs, if they generate predictable output. Simple attacks that exploit insufficient entropy can factor many keys at once, however they are not suitable for targeted attacks, where the victim has a correctly generated key.

Batch GCD and shared prime factors. If two moduli share exactly one prime factor, it can be efficiently determined as their greatest common divisor. When presented with a large set of keys of which some share a prime, a naïve way to find shared factors would be to compute GCD of every pair. A batch GCD algorithm [5] can efficiently compute GCD of millions of RSA moduli. As results of independent researches [29, 20] show, thousands of keys collected on the internet can be factored using this method.

Coppersmith's partial key exposure attack. Coppersmith [13] showed how to find the factors of RSA modulus $n = p \cdot q$, if the $\frac{1}{4} \log_2 n$ most significant bits of p are known. The method has practical applications, as shown in [4]. Smart cards used as national ID cards in Taiwan produced keys, where the upper bits of some primes were predictable due to errors in PRNG.

3 RSA keypair generation in source code and literature

We examined the source codes of 19 open-source cryptographic libraries variants[6] and matched it to the relevant algorithms for primality testing, prime generation and RSA key generation from standards and literature. We then examined how the different methods affected the distributions of the primes and moduli. Summary results for software libraries are available in Table 7. A detailed analysis can be found in [37].

3.1 Prime generation

The choice of the method for generating primes depends on the priorities of the application. Provable primes provide absolute certainty about primality of the number. On the contrary, more efficient algorithms for probable primes can produce a composite number with a very small probability. To prevent some specialized modulus factorization methods, strong primes may be used.

3.1.1 Probable primes

Random numbers (or numbers from a sequence) are tested for primality using probabilistic primality (compositeness) tests. Different libraries use different combinations of the Fermat, Miller-Rabin, Solovay-Strassen and Lucas tests. None of the tests rejects prime numbers if implemented correctly; hence, they do not affect the distribution of the generated primes. GNU Crypto uses a flawed implementation of the Miller-Rabin test. As a result, it permits only Blum primes[7]. No other library generates such primes exclusively (however, some cards do).

In the *random sampling* method, large integers (candidates) are generated until a prime is found. If the candidates are chosen uniformly, the distribution is not biased (case of GNU Crypto 2.0.1, LibTomCrypt 1.17 and WolfSSL 3.9.0).

An *incremental search* algorithm selects a random candidate and then increments it until a prime is found (Botan 1.11.29, Bouncy Castle 1.54, Cryptix 20050328, cryptlib 3.4.3, Crypto++ 5.6.3, FlexiProvider 1.7p7, mbedTLS 2.2.1, SunRsaSign – OpenJDK 1.8.0,

[6]We inspected multiple versions of libraries (though not all exhaustively) to detect code changes relevant to the key generation process. If such a change was detected, both versions were included in the analysis.

[7]A prime p is a Blum prime if $p \equiv 3 \pmod 4$. When both p and q are Blum primes, the modulus n is a Blum integer $n \equiv 1 \pmod 4$.

OpenSSL 1.0.2g, and PGPSDK4). Primes preceded by larger "gaps" will be selected with slightly higher probability; however, this bias is not observable from the distribution of the primes.

Large random integers are likely to have some small prime divisors. Before time-consuming primality tests are performed, compositeness can be revealed through trial division with small primes or the computation of the greatest common divisor (GCD) with a product of a few hundred primes. In the case of incremental search, the sieve of Eratosthenes or a table of remainders that is updated when the candidate is incremented can be used. If implemented correctly, these efficiency improvements do not affect the distribution of the prime generator.

OpenSSL creates a table of remainders by dividing a candidate by small primes. When a composite candidate is incremented, this table is efficiently updated using only operations with small integers. Interestingly, candidates for p for which $p-1$ is divisible by a small prime up to 17 863 (except 2) are also rejected. Such a computational step is useful to speed up the search for a *safe prime*; however, $(p-1)/2$ is not required to be prime by the library (as would be for *safe prime*, see Figure 1). This strange behaviour was first reported by Mironov [34] and can be used to classify the source if the primes are known.

1. Generate random odd candidate c of $nLen/2$ bits.	1. Generate random odd candidate c of $nLen/2$ bits.
2. While c or $c-1$ is divisible by $p_1 \dots p_{2048}$, $c = c + 2$.	2. While c or $c-1$ is divisible by $p_2 \dots p_{2048}$, $c = c + 2$.
3. If c is not a probable prime, go to 1.	3. If c is not a probable prime, go to 1.
4. If $(c-1)/2$ is not a probable prime, go to 1.	4. ~~If $(c-1)/2$ is not a probable prime, go to 1.~~
5. Output a safe prime c.	5. Output an OpenSSL prime c.

Figure 1: *Generating primes using OpenSSL library (right). The algorithm resembles generating safe primes (left), however the primality of* $(p-1)/2$ *is not required by OpenSSL.*

3.1.2 Provable primes

Primes are constructed recursively from smaller primes, such that their primality can be deduced mathematically (using Pocklington's theorem or related facts). This process is randomized; hence, a different prime is obtained each time. An algorithm for constructing provable primes was first proposed by Maurer [32] (used by Nettle 3.2). For each prime p, $p-1$ must have a large factor ($\geq \sqrt{p}$ for Maurer's algorithm or $\geq \sqrt[3]{p}$ for an improved version thereof). Factors of $p+1$ are not affected.

Theorem 3.1 (Pocklington's theorem)**.** *(Citation from [33]). Let* $n \geq 3$ *and let an integer* F *divide* $n-1$*. The prime factorization of* F *is* $F = \prod_{j=1}^{t} q_j^{e_j}$*. If there exist an integer* a *satisfying:*

- $a^{n-1} \equiv 1 \bmod n$*; and*
- $\mathsf{GCD}(a^{(n-1)/q_j} - 1, n) = 1$ *for each* j*,* $i \leq j \leq t$*,*

then every prime divisor p *of* n *is congruent to 1 modulo* F*. If* $F > \sqrt{n} - 1$*, then* n *is prime.*

3.1.3 Strong primes

A prime p is *strong* if both $p-1$ and $p+1$ have a large prime factor (used by libgcrypt 1.65 in FIPS mode and by the OpenSSL 2.0.12 FIPS module). We also refer to these primes as *FIPS-compliant*, as FIPS 186-4 requires such primes for 1024-bit keys (larger keys may use probable primes). Differing definitions of strong primes are given in the literature; often, the large factor of $p-1$ itself (minus one) should also have a large prime factor (PGPSDK4 in FIPS mode). Large random primes are not "weak" by comparison, as their prime factors are sufficiently large, with sufficient probability, to be safe from relevant attacks.

Strong primes are constructed from large prime factors. They can be generated uniformly (as in ANSI X9.31, FIPS 186-4, and IEEE 1363-2000) or with a visibly biased distribution (as in a version of Gordon's algorithm [17] used in PGPSDK4).

3.2 Key generation – prime pairs

The key size is the bit length of the modulus. Typically, an algorithm generates keys of an exact bit length (the only exception being PGPSDK4 in FIPS mode). The primes are thus generated with a size equal to half of the modulus length. These two measures define the *maximal region* for RSA primes. The product of two k-bit primes is either $2k$ or $2k-1$ bits long. There are two principal methods of solving the problem of short $(2k-1)$-bit moduli, as illustrated in Figure 2.

3.2.1 Rejection sampling

Pairs of k-bit primes are generated until their product has the correct length. To produce an unbiased distribution, two new primes should be generated each time (Cryptix 20050328, FlexiProvider 1.7p7, and mbedTLS 2.2.1). If the greater prime is kept and only one new prime is generated, some bias can be observed in the resulting distribution of

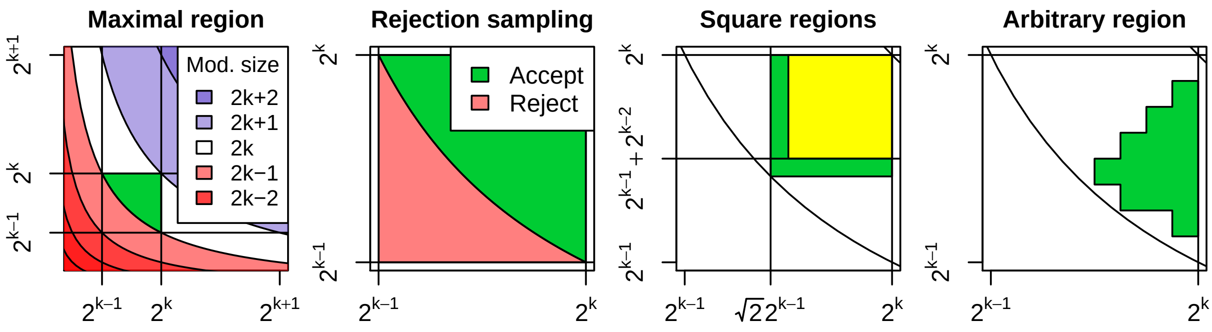


Figure 2: *RSA key generation. The maximal region for the generated primes is defined by the precise length of the modulus and equal lengths of the primes. Such keys can be generated through rejection sampling. To avoid generating short moduli (which must be discarded), alternative square regions may be used. Several implementations, such as that of the NXP J2A080 card, generate primes from arbitrary combinations of square regions.*

RSA moduli (Bouncy Castle up to version 1.53 and SunRsaSign in OpenJDK 1.8.0). If the first prime is kept (without regard to its size) and the second prime is re-generated, small moduli will be much more probable than large values (GNU Crypto 2.0.1).

3.2.2 “Square” regions

This technique avoids moduli of incorrect length by generating only larger primes, whose product has the correct length. Typically, both primes are selected from identical intervals, hence producing a square region when plotted in two dimensions.

The smallest k-bit numbers that produce a 2k-bit modulus are close to $\sqrt{2}\cdot 2^{k-1}$. Since random numbers can be easily generated uniformly from intervals bounded by powers of two, the distribution must be additionally transformed to fit such an interval. We refer to the prime pairs generated from the interval $\left[\sqrt{2}\cdot 2^{k-1}, 2^k - 1\right]$ as coming from the *maximal square region* (Bouncy Castle since 1.54, Crypto++ 5.6.3, and the Microsoft cryptography providers used in CryptoAPI, CNG and .NET). Crypto++ approximates this interval by generating the most significant byte of primes from 182 to 255.

A more *practical square region*, which works well for candidates generated uniformly from intervals bounded by powers of two, is achieved by fixing the two most significant bits of a candidate to 11_2 (Botan 1.11.29, cryptlib 3.4.3, libgcrypt 1.6.5, LibTomCrypt 1.17, OpenSSL 1.0.2g, PGPSDK4, and WolfSSL 3.9.0). Additionally, the provable primes generated in Nettle 3.2 and the strong primes generated in libgcrypt 1.6.5 (in FIPS mode) and in the OpenSSL 2.0.12 FIPS module are produced from this region.

4 Analysis of the generated RSA key pairs

The key pairs extracted from both the software libraries and the cards were examined using a similar set of analytical techniques. The goal was to identify sources with the same behaviour, investigate the impact on the public key values and infer the probable key generation algorithm used based on similarities and differences in the properties.

4.1 Distributions of the primes

To visualize the regions from which pairs of primes were chosen, we plotted the most significant byte (MSB) of each prime on a heat map. It is possible to observe the *intervals for prime generation*, as discussed in Section 3.

Figures 3 to 10 show all the observed distributions. Surprisingly, the MSB patterns were significantly different for the cards and the software implementations. The patterns were identical among different physical cards of the same type and were also shared between some (but not all) types of cards from the same manufacturer (probably because of a shared code base). The most peculiar card distributions can be found in Figure 10. We did not encounter any library that produced outputs comparable to the cards Infineon JTOP 80K or Gemalto GXP E64. The output of NXP cards could be reproduced by generating primes alternately and uniformly from 14 different regions, each characterized by a pattern in the top four bits of the primes. By comparison, it was rarer for a bias to be introduced by a library.

The relation between the values of p and q reveals additional conditions placed on the primes, such as a *minimal size of the difference* $p - q$ (PGPSDK4, NXP J2D081, and NXP J2E145G).

It is possible to verify whether *small factors of* $p - 1$ are being avoided (e.g., OpenSSL or NXP J2D081) or whether the primes generally do not exhibit same distribution as randomly generated numbers (Infineon JTOP 80K) by computing the distributions of the primes, modulo small primes. It follows from Dirichlet's theorem that the remainders should be distributed uniformly among the $\phi(n)$ congruence classes in $\mathbb{Z}_n^*$ [33, Fact 4.2].

The patterns observed for the 512-bit keys were found to be identical to those for the stronger keys of 1024 and 2048 bits. For the software implementations, we checked the source codes to confirm that there were no differences in the algorithms used to generate keys of different lengths. For the cards, we assume the same and generalize the results obtained for 512-bit RSA keys to the longer (and much more common) keys.

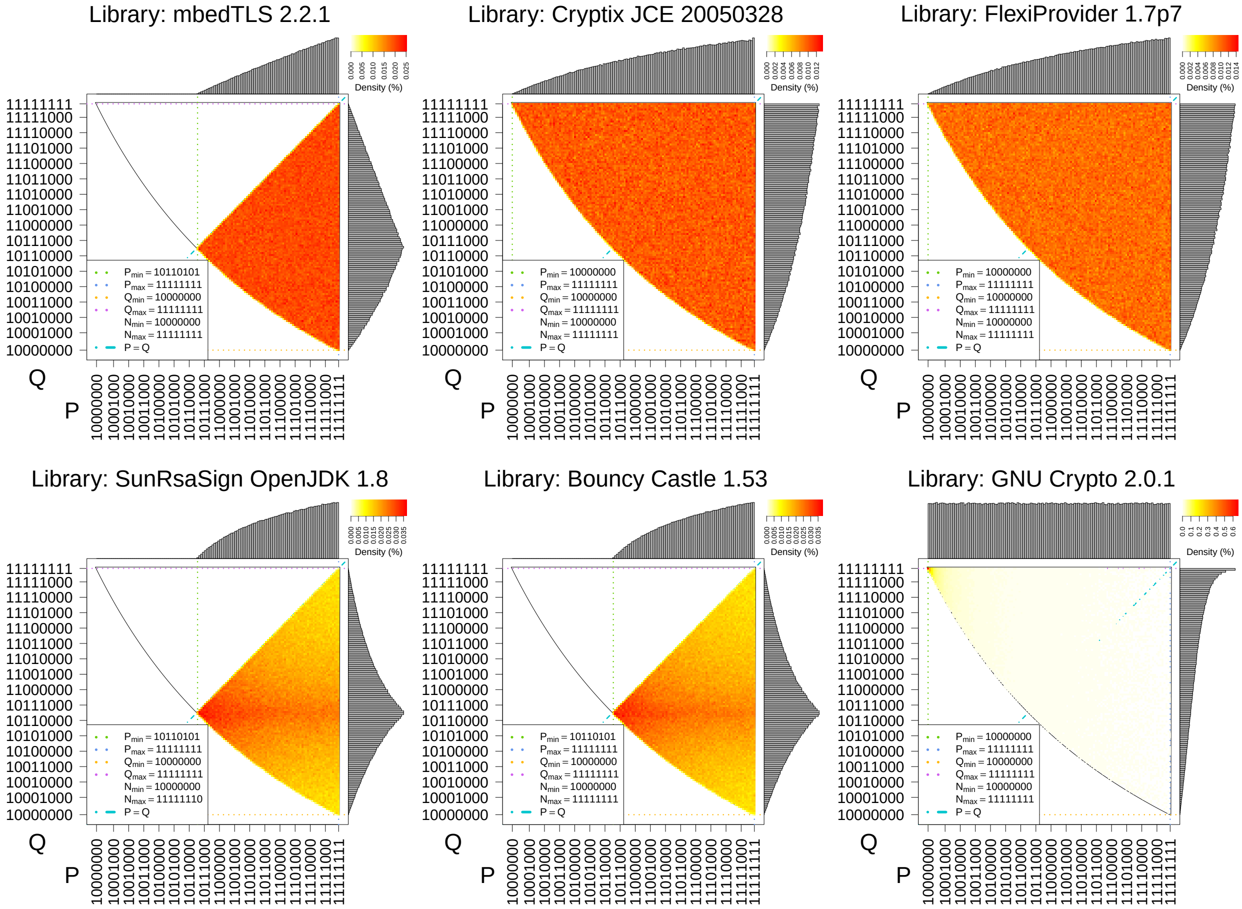


Figure 3: *Libraries: rejection sampling. The histograms on the top and the side of the graph represent the marginal distributions of* p *and* q*, respectively. The colour scheme expresses the likelihood that primes of a randomly generated key will have specific high order bytes, ranging from white (not likely) over orange to red (more likely). The libraries mbedTLS 2.2.1, Cryptix JCE 20050328 and FlexiProvider 1.7p7 generate a fresh pair of primes, if their product has incorrect bit length. SunRsaSign (default provider in OpenJDK 1.8) and Bouncy Castle 1.53 keep the larger prime. GNU Crypto 2.0.1 keeps the first generated prime.*

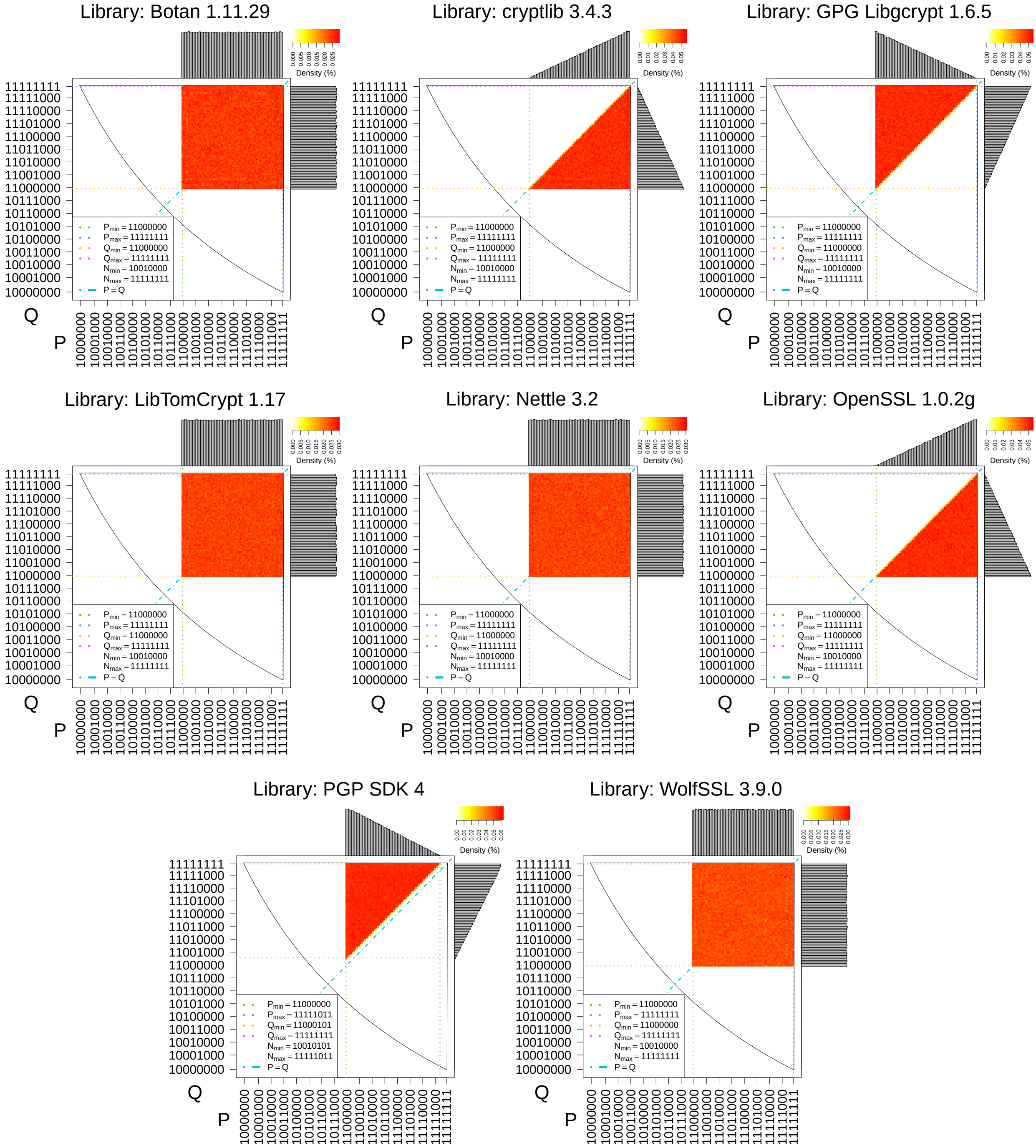


Figure 4: *Libraries: the practical square region. OpenSSL 1.0.2g and cryptlib 3.4.3 order the primes after generation such that* $p > q$*, according to a convention for CRT keys. Libgcrypt 1.6.5 (used by GPG) and PGP SDK 4 sort the primes in the opposite order (*$p < q$*). Other libraries do not manipulate with the order of the primes. The graphs also reveals that PGP SDK 4 requires that the primes differ somewhere in their top 6 bits. Nettle 3.2 generates provable primes, however this type of graph does not help to distinguish the type of the prime generator. See Figure 3 for the graph interpretation.*

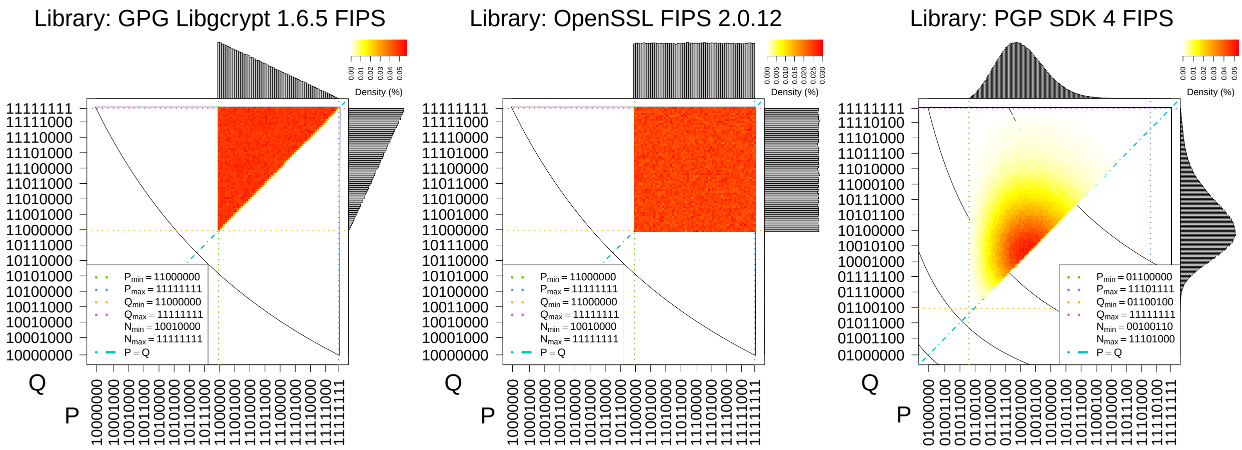


Figure 5: *Libraries: FIPS variants. Libgcrypt 1.6.5 in FIPS mode and OpenSSL FIPS module 2.0.12 generate strong primes uniformly and target the practical square region. PGP SDK 4 in FIPS mode generate strong primes from a different distribution. The primes may be one bit shorter than expected (notice the different values at the axes). As a result, the modulus may be shorter than requested, by one or two bits. See Figure 3 for the graph interpretation.*

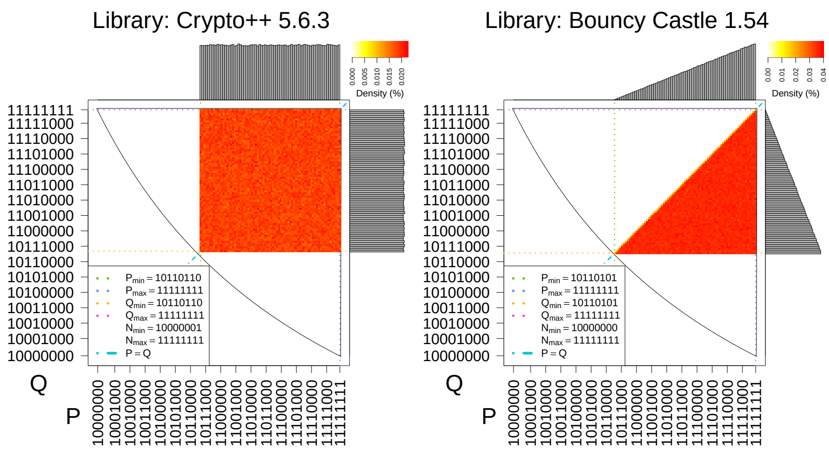


Figure 6: *Libraries: the maximal square region. In the case of Crypto++ 5.6.3, the interval for prime generation is approximated. Bouncy Caste 1.54 generates primes from the truly maximal square region (besides ordering the primes afterwards). See Figure 3 for the graph interpretation.*

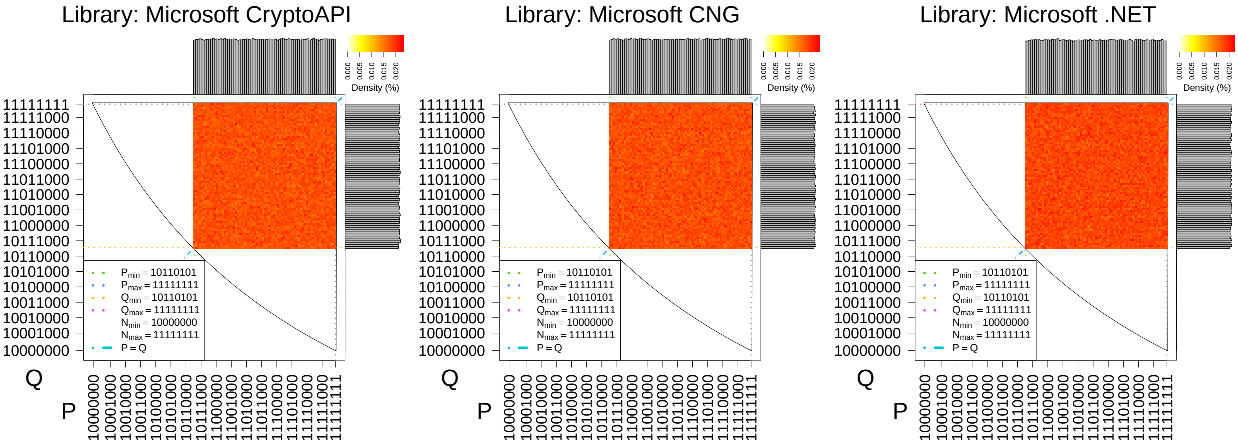


Figure 7: *Black-box software implementations. Microsoft libraries (CryptoAPI, CNG and the default provider in .NET) seem to be using the same parameters for prime intervals – the maximal square region. Further analysis revealed, that the uniformly generated primes are strong. This type of graph does not capture the type of generated primes. See Figure 3 for the graph interpretation.*

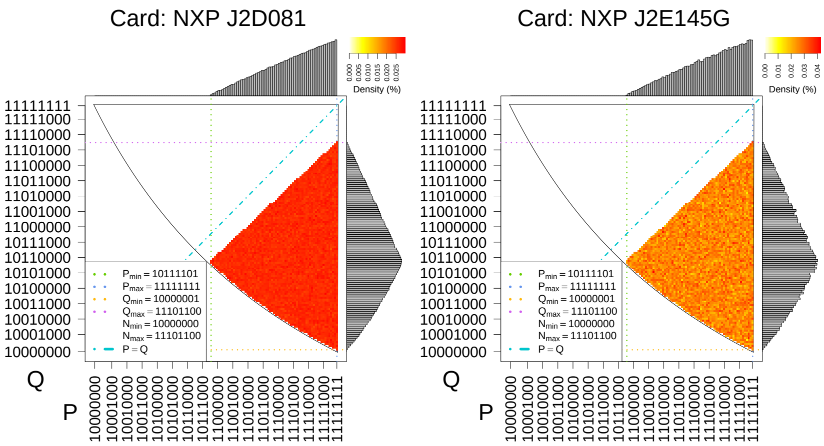


Figure 8: *Cards: possibly rejection sampling. The primes generated by NXP J2D081 and NXP J2E145G are sorted and generated almost from the maximal region defined by modulus and prime bit lengths. Additionally, some requirement is placed on the difference of the primes. We hypothesize that the distribution can be easily reproduced by rejection sampling. See Figure 3 for the graph interpretation.*

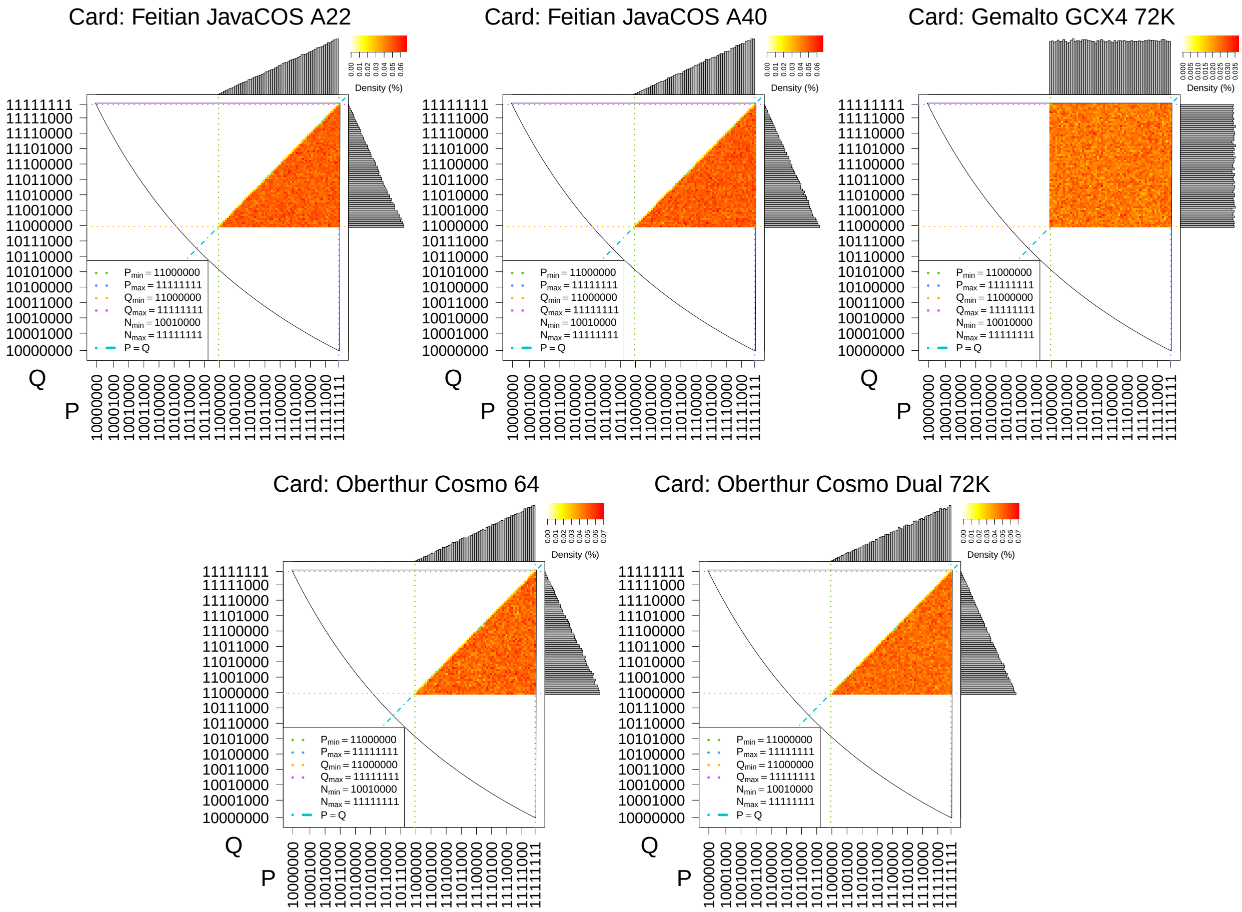


Figure 9: *Cards: the practical square region. Gemalto GCX4 72K generates strong primes uniformly, however the difference in the graphs is caused by ordering the primes at the end of key generation, not by type of primes. See Figure 3 for the graph interpretation.*

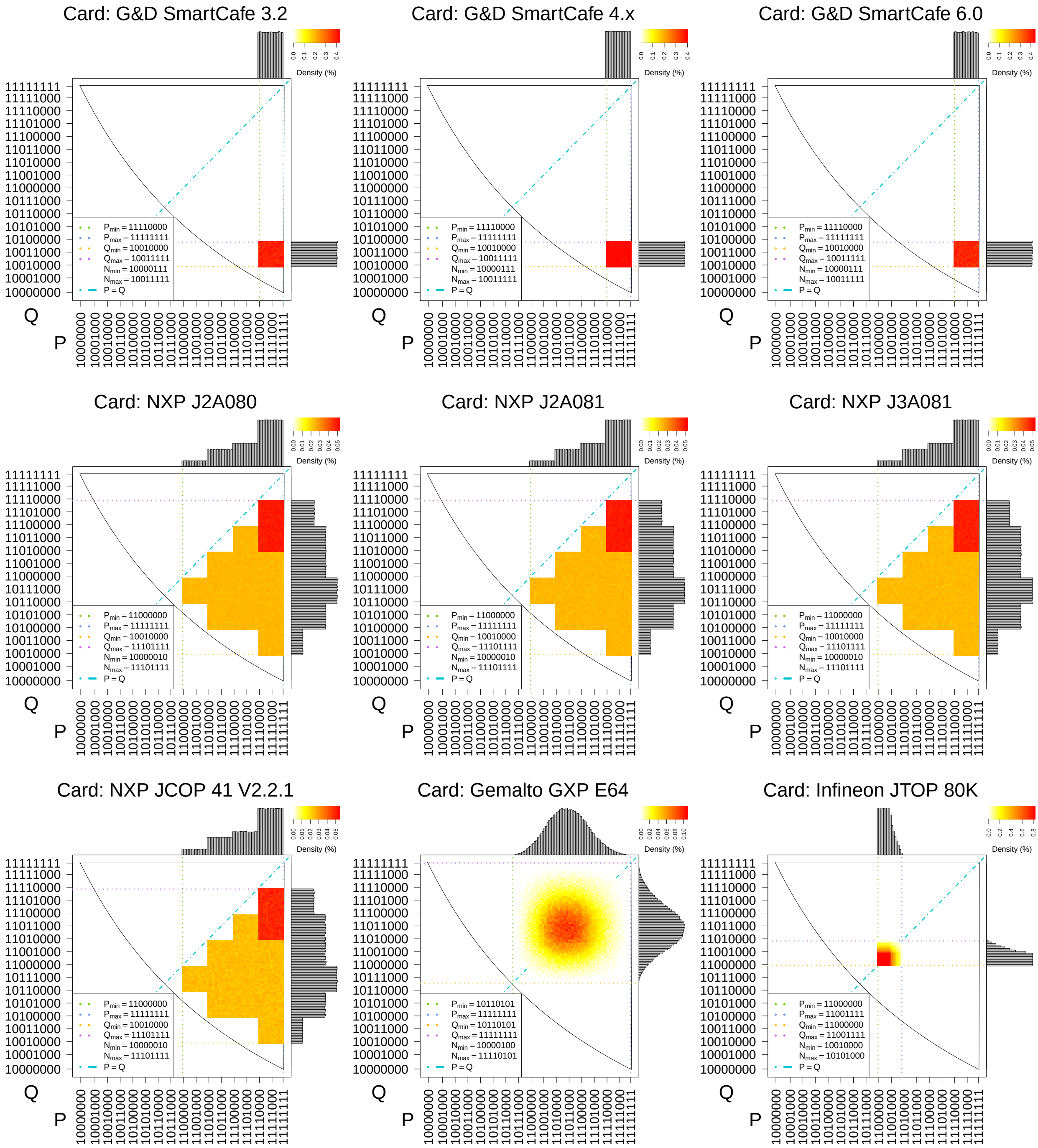


Figure 10: *Cards: arbitrary combination of square regions and implementation specific regions. G&D cards generate primes from a square region, defined by patterns in the top four bits of the primes –* 1111_2 *and* 1001_2 *for* p *and* q*, respectively. Some NXP cards generate the primes from 14 regions defined by distinct pairs of patterns in the top four bits of the primes. Two such regions are twice as probable as others. Gemalto GXP E64 generates strong primes, the distribution is not uniform. Infineon JTOP 80K has a proprietary algorithm for key generation, which has several characteristics which are unique in our dataset. See Figure 3 for the graph interpretation.*

4.2 Distributions of the moduli

The MSB of a modulus is directly dependent on the MSBs of the corresponding primes p and q. As seen in Figures 11 and 12, if an observable pattern exists in the distributions of the MSBs of primes p and q, a noticeable pattern also appears in the MSB of the modulus. The preservation of shared patterns was observed for all tested types of software libraries and cards. The *algorithm used for prime pair selection* can often be observed from the distribution of the moduli. If a source uses an atypical algorithm, it is possible to detect it with greater precision, even if we do not know the actual method used.

Non-randomness with respect to *small factors of* $p - 1$ can also be observed from the modulus, especially for small divisors. Whereas random primes are equiprobably congruent to 1 and 2 modulo 3 (Table 2), OpenSSL primes are always congruent to 2 (Table 3). As a result, an OpenSSL modulus is always congruent to 1 modulo 3 (Table 4). This property is progressively more difficult to detect for larger prime divisors. The moduli are more probably congruent to 1 modulo all small primes, which are avoided from $p-1$ by OpenSSL. However, the bias is barely noticeable for prime factors of 19 and more, even in an analysis of a million keys. OpenSSL primes are congruent to 1 modulo 5 with probability 1/3 (as opposed to 1/4 for random primes, Table 5), to 1 modulo 7 with probability 1/5 (as opposed to 1/6), and to 1 modulo 11 with probability 1/9 (as opposed to 1/10). For the practical classification of only a few keys (see Section 5), we use only the remainder of division by 3.

If a source has a characteristic distribution of moduli modulo small primes, as is the case of the Infineon JTOP 80K smartcard (Table 6), it can be used for a more precise classification. However, in our general case, the single source is relatively scarce and no other sources behave similarly, therefore we have decided not to use the distributions observed for this card as a classification trait.

The use of *Blum integers* can also be detected from the moduli with a high precision, as random moduli are equiprobably congruent to 1 and 3 modulo 4, whereas Blum integers are always congruent to 1 modulo 4. The probability that k random moduli will be Blum integers is 2^{-k}.

Neither libraries nor cards attempt to achieve a uniform distribution of moduli. Existing algorithms [21, 31] have the disadvantage that sometimes a prime will be one bit larger than half of the modulus length. All sources sacrifice uniformity in the most significant bits of the modulus to benefit from more efficient methods of prime and key generation.

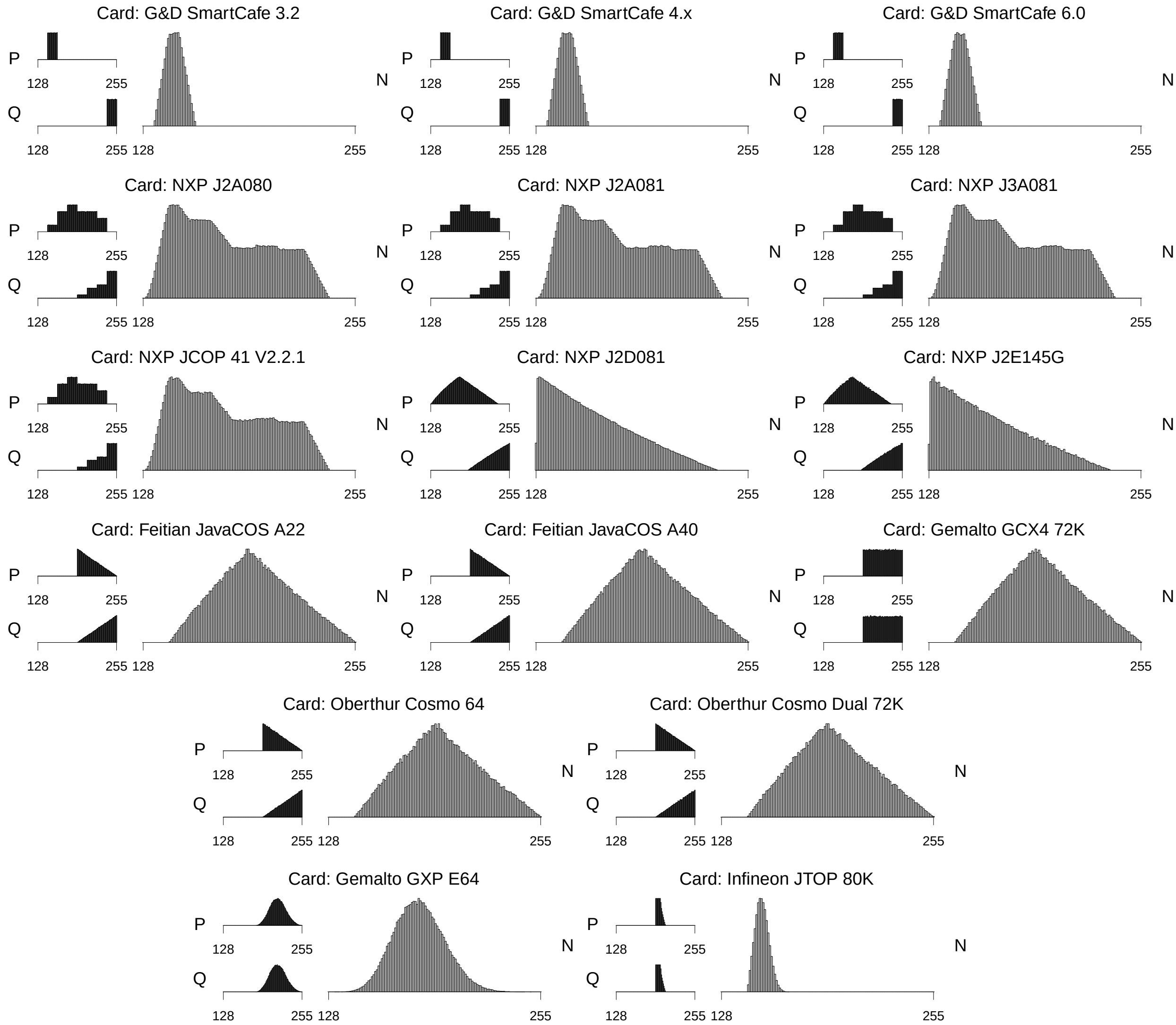


Figure 11: *Cards: the effect of distribution of the MSB of primes on the distribution of the MSB of the modulus.*

We verified that the distribution of the other bytes of the moduli is otherwise uniform. The second least significant bit is biased in the case of Blum integers. Sources that use the same algorithm are not mutually distinguishable from the distributions of their moduli.

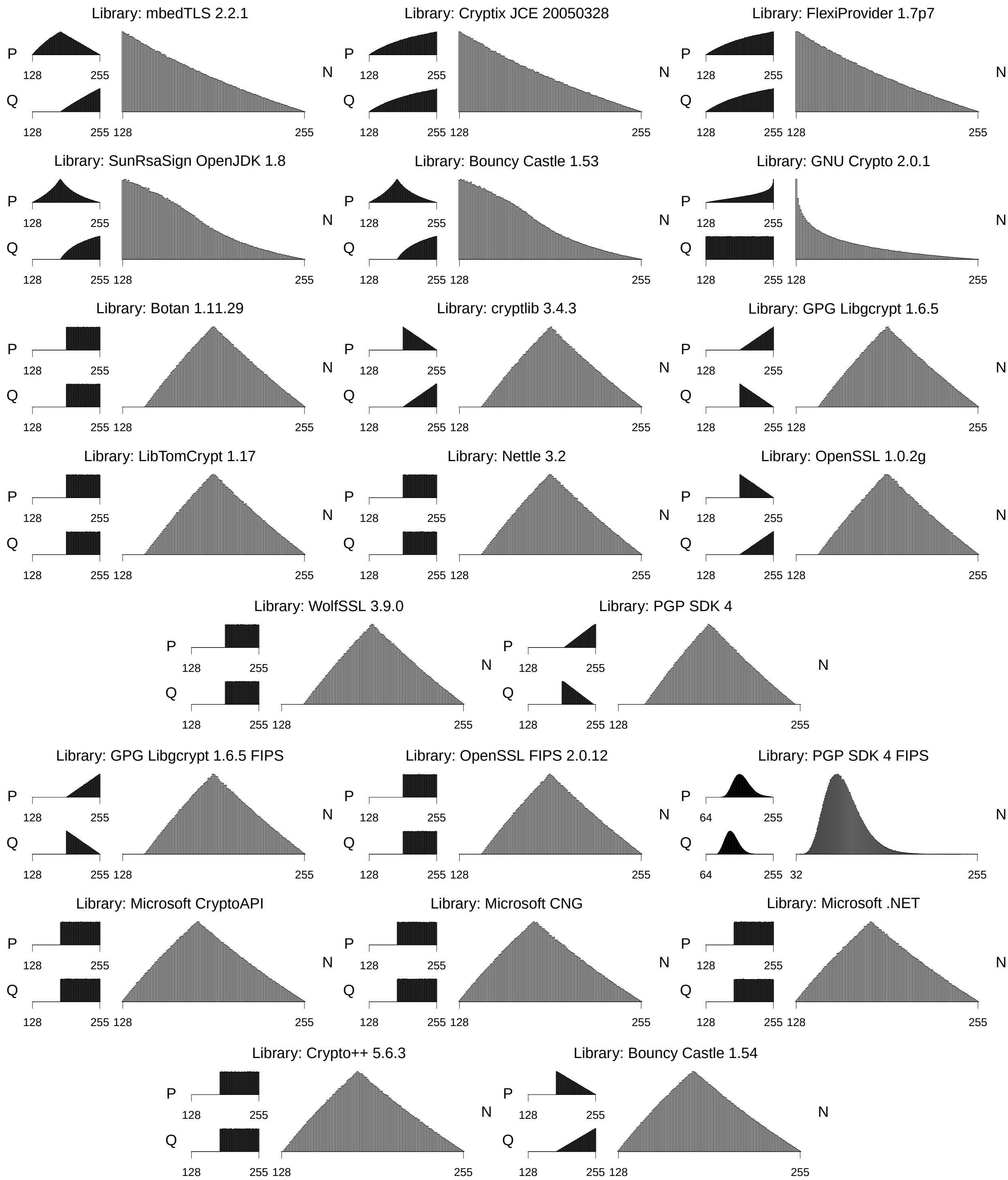


Figure 12: *Software libraries: the effect of distribution of the MSB of primes on the distribution of the MSB of the modulus.*

	Remainder																	
Divisor	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	1/2	1/2																
5	1/4	1/4	1/4	1/4														
7	1/6	1/6	1/6	1/6	1/6	1/6												
11	1/10	1/10	1/10	1/10	1/10	1/10	1/10	1/10	1/10	1/10								
13	1/12	1/12	1/12	1/12	1/12	1/12	1/12	1/12	1/12	1/12	1/12	1/12						
17	1/16	1/16	1/16	1/16	1/16	1/16	1/16	1/16	1/16	1/16	1/16	1/16	1/16	1/16	1/16	1/16		
19	1/18	1/18	1/18	1/18	1/18	1/18	1/18	1/18	1/18	1/18	1/18	1/18	1/18	1/18	1/18	1/18	1/18	1/18

Table 2: *Probability of remainders modulo small primes for uniformly generated large prime. Same probabilities apply to remainders for the public key (modulus $n = p \cdot q$), if primes are chosen uniformly and independently.*

	Remainder																	
Divisor	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	0	1																
5	0	1/3	1/3	1/3														
7	0	1/5	1/5	1/5	1/5	1/5												
11	0	1/9	1/9	1/9	1/9	1/9	1/9	1/9	1/9	1/9								
13	0	1/11	1/11	1/11	1/11	1/11	1/11	1/11	1/11	1/11	1/11	1/11						
17	0	1/15	1/15	1/15	1/15	1/15	1/15	1/15	1/15	1/15	1/15	1/15	1/15	1/15	1/15	1/15		
19	0	1/17	1/17	1/17	1/17	1/17	1/17	1/17	1/17	1/17	1/17	1/17	1/17	1/17	1/17	1/17	1/17	1/17
…																		
17 863	0	1/17 861	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…
17 881	1/17 880	1/17 880	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…
17 891	1/17 890	1/17 890	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…

Table 3: *Probability of remainders modulo small primes for primes generated by OpenSSL.*

	Remainder																
Divisor	1	2	3	4	5	6	7	8	9	10	11	12	…	16	…	18
3	1	0														
5	3/9	2/9	2/9	2/9												
7	5/25	4/25	4/25	4/25	4/25	4/25										
11	9/81	8/81	8/81	8/81	8/81	8/81	8/81	8/81	8/81	8/81						
13	11/121	10/121	10/121	10/121	10/121	10/121	10/121	10/121	10/121	10/121	10/121	10/121				
17	15/225	14/225	14/225	14/225	14/225	14/225	14/225	14/225	14/225	14/225	14/225	14/225	…	14/225		
19	17/289	16/289	16/289	16/289	16/289	16/289	16/289	16/289	16/289	16/289	16/289	16/289	…	16/289	…	16/289
…																
17 863	$\frac{1}{17\,861}$	$\frac{17\,860}{17\,861^2}$	…	…	…	…	…	…	…	…	…	…	…	…	…	…
17 881	1/17 880	1/17 880	…	…	…	…	…	…	…	…	…	…	…	…	…	…
17 891	1/17 890	1/17 890	…	…	…	…	…	…	…	…	…	…	…	…	…	…

Table 4: *Probability of remainders modulo small primes for moduli generated by OpenSSL.*

	p mod 5			
q mod 5	1	2	3	4
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

	p mod 5			
q mod 5	1	2	3	4
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

Table 5: *The remainder modulo a small prime* q *(the example is using* $q = 5$*) when dividing product of two primes with specific remainders. If the primes with remainder 1 are avoided (i.e.* $p - 1$ *is not divisible by the small prime), the modulus is congruent to 1 with probability* $(q-2)/(q-2)^2$*, while the other remainders are less probable* $(q-3)/(q-2)^2$*.*

	Remainder																							
Divisor	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	...	22	...	26	...	36
3	1/2	1/2																						
5	1/4	1/4	1/4	1/4																				
7	1/6	1/6	1/6	1/6	1/6	1/6																		
11	1/2	0	0	0	0	0	0	0	0	1/2														
13	1/6	0	1/6	1/6	0	0	0	0	1/6	1/6	0	1/6												
17	1/8	1/8	0	1/8	0	0	0	1/8	1/8	0	0	0	1/8	0	1/8	1/8								
19	1/9	0	0	1/9	1/9	1/9	1/9	0	1/9	0	1/9	0	0	0	0	1/9	1/9	0						
23	1/22	1/22	1/22	1/22	1/22	1/22	1/22	1/22	1/22	1/22	1/22	1/22	1/22	1/22	1/22	1/22	1/22	1/22	...	1/22				
29	1/28	1/28	1/28	1/28	1/28	1/28	1/28	1/28	1/28	1/28	1/28	1/28	1/28	1/28	1/28	1/28	1/28	1/28	...	1/28	...	1/28	...	
31	1/30	1/30	1/30	1/30	1/30	1/30	1/30	1/30	1/30	1/30	1/30	1/30	1/30	1/30	1/30	1/30	1/30	1/30	...	1/30	...	1/30	...	
37	1/3	0	0	0	0	0	0	0	0	1/3	0	0	0	0	0	0	0	0	...	0	...	1/3	...	0

Table 6: *Probability of remainders modulo small primes for primes and moduli generated by Infineon JTOP 80K smartcard. Additionally, we detected that the remainders modulo 53, 61, 71, 73, 79, 97, 103, 107, 109, 127, 151 and 157 are also from certain subgroups of residue classes and do not represent all residue classes. The values are almost uniformly distributed modulo all other primes (tested up to 547), as expected from the Dirichlet's theorem. We use the remainder modulo 3 for our classification. The card generates uniformly distributed moduli modulo 3, therefore the criterion is not applicable to this card. However, if we would focus more on this source in our analysis, by using the remainders modulo other primes, the card could be identified much more easily. Hence for specialized cases other criteria can be more useful than for our general analysis.*

4.3 Factorization of $p-1$ and $p+1$

It is possible to verify whether *strong primes* are being used. Most algorithms generate strong primes from uniform distributions (ANSI X9.31, FIPS 186-4, IEEE 1363, OpenSSL FIPS, libgcrypt FIPS, Microsoft and Gemalto GCX4 72K), matching the distribution of random primes, although PGPSDK4 FIPS produces a highly atypical distribution of primes and moduli, such that this source can be detected even from public keys. Hence, we were obliged to search for the sizes of the prime factors of $p-1$ and $p+1$ directly[8] by factoring them using the YAFU software package [53]. We then extended the results obtained for 512-bit keys to the primes of 1024-bit key pairs (though based on fewer factorized values because of the longer factorization time). Finally, we extrapolated the results to keys of 2048 bits and longer based on the known patterns for shorter keys.

As confirmed by the source code, large factors of $p \pm 1$ generated in OpenSSL FIPS and by libgcrypt in FIPS mode always have 101 bits; this value is hardcoded. PGPSDK4 in FIPS mode also generates prime factors of fixed length; however, their size depends on the size of the prime.

Additionally, we detected strong primes in some of our black-box sources. Gemalto GCX4 72K generates strong primes uniformly, but the large prime factors always have 101 bits. The strong primes of Gemalto GXP E64, which have 112 bits, are not drawn from a uniform distribution. The libraries that use Microsoft cryptography providers (CryptoAPI, CNG, and .NET) produce prime factors of randomized length, ranging from 101 bits to 120 bits, as required by ANSI X9.31.

For large primes, $p \pm 1$ has a large prime factor with high probability. A random integer p will not have a factor larger than $p^{1/u}$ with a probability of approximately u^{-u} [33]. Approximately 10% of 256-bit primes do not have factors larger than 100 bits, but 512-bit keys are not widely used. For 512-bit primes, the probability is less than 0.05%. Therefore, the requirement of a large factor does not seem to considerably decrease the number of possible primes. However, many sources construct strong primes with factors of exact length (e.g., 101 bits). Using the approximation of the prime-counting function $\pi(n) \approx \frac{n}{\ln(n)}$ [33], we estimate that the interval required by ANSI X9.31 (prime factors from 101 to 120 bits) contains approximately 2^{20} times more primes than the number of 101-bit primes. Hence, there is a loss of entropy when strong primes are generated in this way, although we are not aware of an attack that would exploit this

[8]By $p-1$, we always refer to both $p-1$ and $q-1$, as we found no relevant difference between p and q in the factorization results.

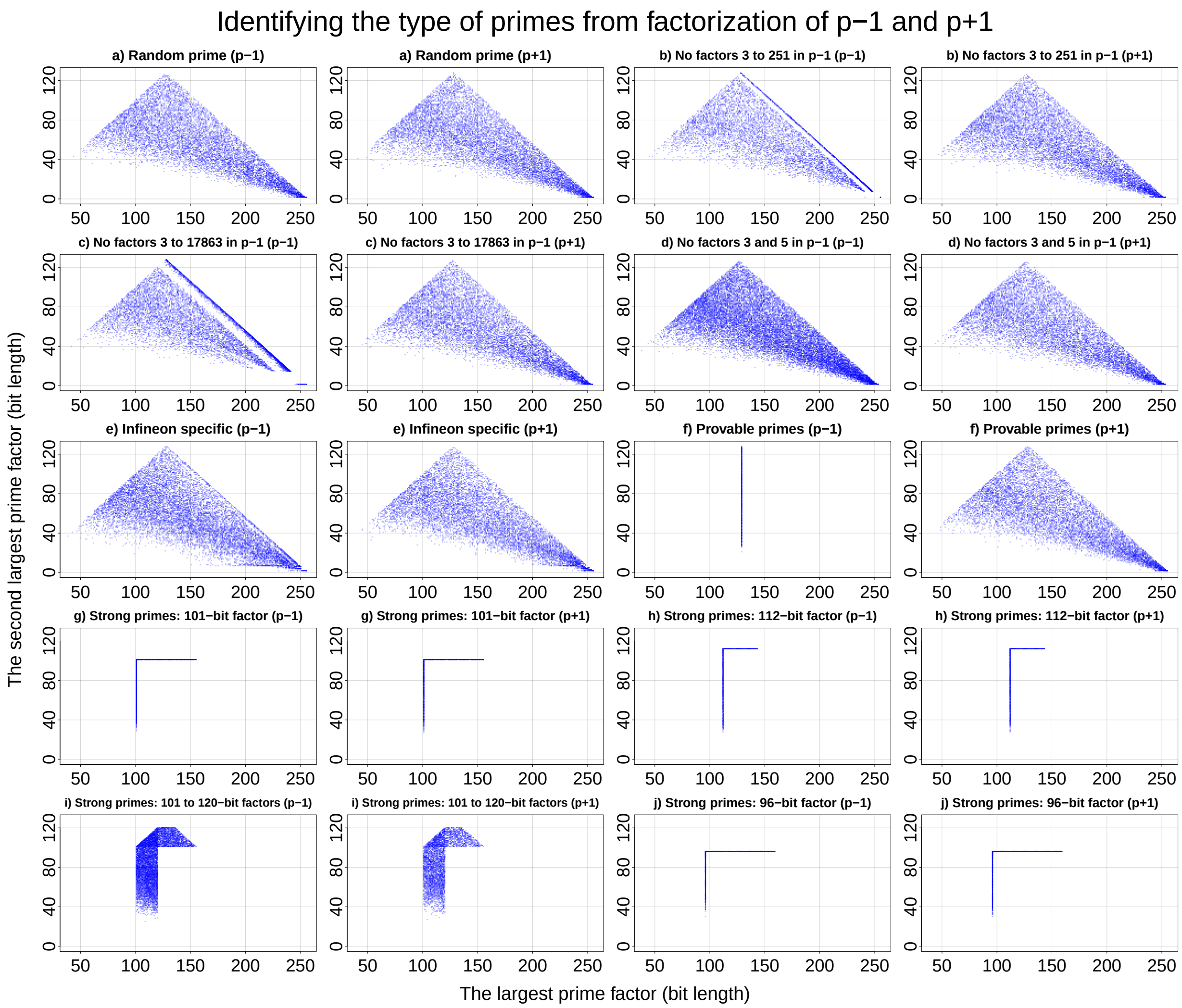


Figure 13: *Identifying the type of primes from scatter graphs of two biggest factors of* $p - 1$ *and* $p + 1$ *for 512-bit RSA. The tested sources fall into following categories:* ***a) Random primes:*** *Botan 1.11.29, Bouncy Castle 1.53 & 1.54, Cryptix JCE 20050328, cryptlib 3.4.3, Crypto++ 5.6.3, FlexiProvider 1.7p7, GNU Crypto 2.0.1, (GPG) libgcrypt 1.6.5, LibTomCrypt 1.17, mbedTLS 2.2.1, PGPSDK4, SunRsaSign (OpenJDK 1.8), G&D SmartCafe 3.2, Feitian JavaCOS A22, Feitian JavaCOS A40, NXP J2A080, NXP J2A081, NXP J3A081, NXP JCOP 41 V2.2.1, Oberthur Cosmo Dual 72K;* ***b) No factors 3 to 251 in p-1:*** *NXP J2D081, NXP J2E145G;* ***c) No factors 3 to 17863 in p-1:*** *OpenSSL 1.0.2g;* ***d) No factors 3 and 5 in p-1:*** *G&D SmartCafe 4.x, G&D SmartCafe 6.0;* ***e) Infineon specific:*** *Infineon JTOP 80K;* ***f) Provable primes:*** *Nettle 3.2;* ***g) Strong primes with 101-bit prime factors of*** $p \pm 1$***:*** *(GPG) libgcrypt 1.6.5 FIPS, OpenSSL FIPS 2.0.12, Gemalto GCX4 72K;* ***h) Strong primes with 112-bit prime factors of*** $p \pm 1$***:*** *Gemalto GXP E64;* ***i) Strong primes with 101 to 120-bit prime factors of*** $p \pm 1$***:*** *MS CNG, MS CryptoAPI, MS .NET;* ***j) Strong primes with 96-bit prime factors of*** $p \pm 1$***:*** *PGPSDK4 FIPS.*

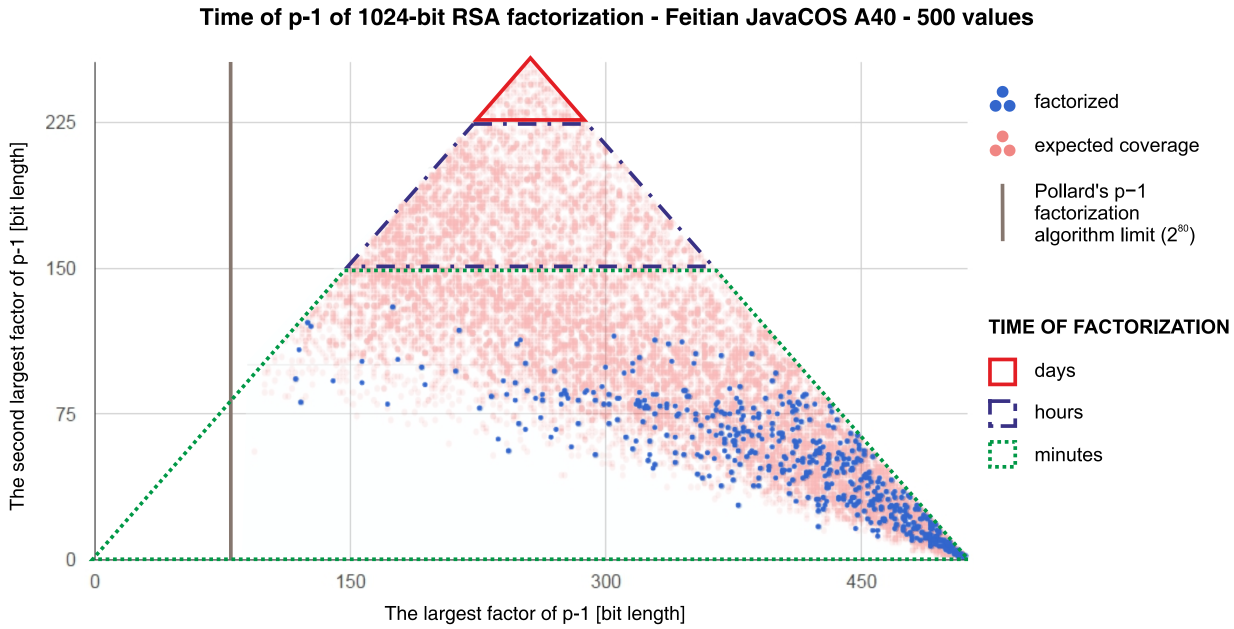


Figure 14: *Scatter graph of dependency of lengths of the largest and the second largest factor of* $p - 1$ *for 1024-bit RSA for the Feitian JavaCOS A40 card. Blue dots represent several hundred actually factorized values overlayed on expected results interpolated from the distribution of factors for 512-bit RSA (where 10 000 values were taken and all factorized). The factorized values of* $p - 1$ *from 1024-bit RSA fit into projected area. The triangle in the upper center without any actual factors found is due to a time limit we allowed to be spent on the factorization of a single value. The factorization of* $p - 1$ *which is expected to fall into given area requires from 500 to 2 000 hours of factorization time.*

fact. For every choice of an auxiliary prime, 2^{93} possible values are considered instead of 2^{113}, which implies the loss of almost 20 bits of entropy. If the primes are to be (p^-, p^+)-safe, then 2 auxiliary primes must be generated. Because we require two primes p and q for every RSA key, we double the estimated loss of entropy compared with ANSI-compliant keys to 80 bits for 1024-bit keys.

When $p - 1$ is guaranteed to have a large prime factor but $p + 1$ is not, the source is most likely using *provable primes*, as in the case of the Nettle library. Techniques for generating provable primes construct p using a large prime factor of $p - 1$ (at least $\sqrt{p}$ for Maurer's algorithm or $\sqrt[3]{p}$ for an improved version thereof). The size of the prime factors of $p + 1$ is not affected by Maurer's algorithm.

Factorization also completes the picture with regard to the avoidance of small factors in $p - 1$. Sources that avoid small factors in $p - 1$ achieve a smaller number of factors on average (and therefore also a higher average length of the largest factor). No small

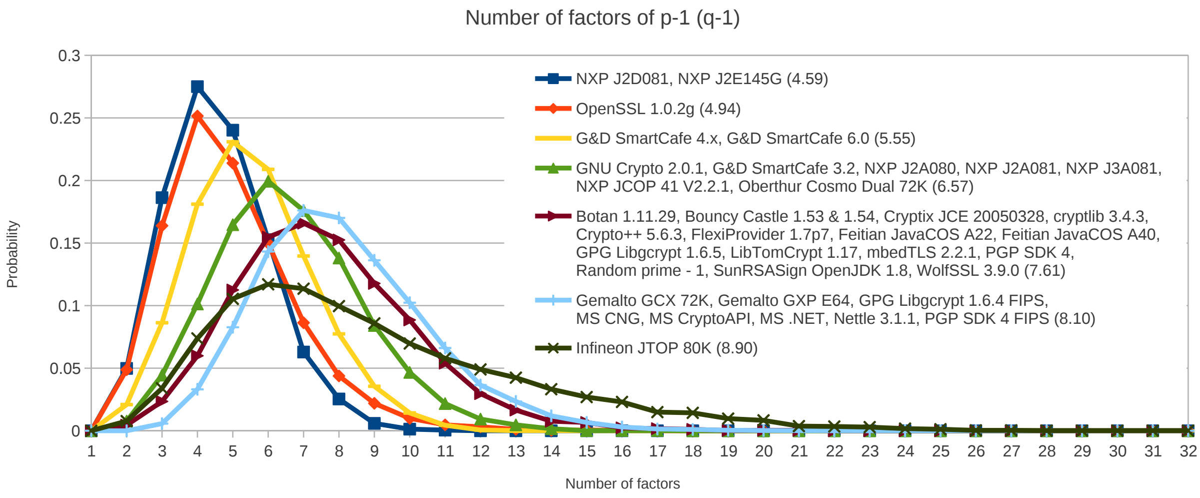


Figure 15: *Number of factors of* $p - 1$ *for RSA 512 bits. Histogram is generated from 10 000 factorized values of* $p - 1$ *and* $q - 1$ *taken from 5 000 key pairs. We identified 7 groups based on similar distributions. The average number of factors is listed at the end of each group. The first 3 groups with the least average number of factors are avoiding small factors in* $p - 1$*, therefore increasing the average size of the factor and decreasing the expected number of factors. Values generated from a large portion of the libraries behave the same way as a prime randomly generated by incremental search, decreased by one. Two other groups have similar distributions. Sources that generate strong primes or provable primes belong to a separate category as well. The implementation choices of Infineon JTOP 80K card lead to large number of small factors, resulting in high number of factors (8.90 on average, with as many as 32 factors observed).*

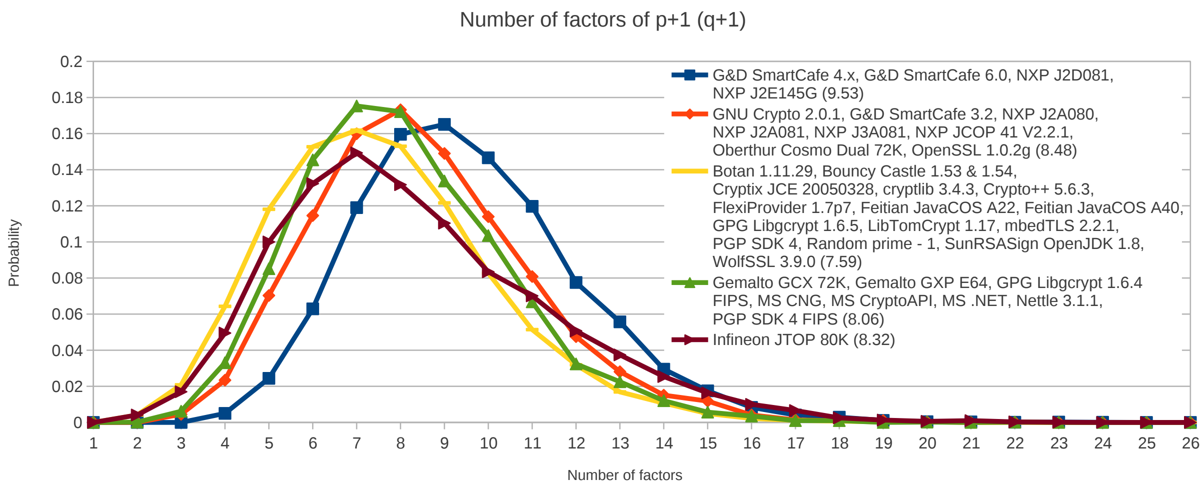


Figure 16: *Number of factors of* $p + 1$ *for RSA 512 bits. Histogram is generated from 10 000 factorized values of* $p + 1$ *and* $q + 1$ *taken from 5 000 key pairs. The sources behave more consistently, with only 5 distinguishable categories observed.*

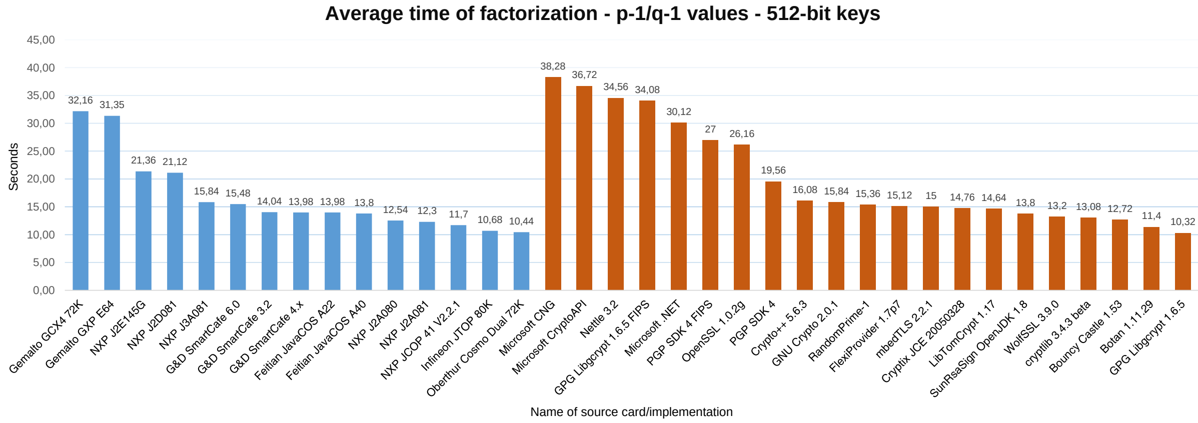


Figure 17: *Average factorization time of* $p-1$ *values with YAFU factorization tool (methods used: trial division, Pollard rho, ECM, Pollard's p-1, Williams p+1, Fermat, Self-initializing quadratic sieve based on suitability) on CPU Intel i5 M430 @ 2x2.27 GHz (6GB RAM available, but only about 100MB used). For every source, 1000 values were factorized. Since the factorization of* $p-1$ *requires the knowledge of the private key, the measured values have no direct connection with the strength of the keys. However, we observed that* $p-1$ *values coming from strong primes (always having a large factor) were the most difficult to factor.*

factors are present in keys from NXP J2D081 and J2E145G (values from 3 to 251 are avoided), from OpenSSL (values from 3 to 17 863 are avoided) and from G&D Smartcafe 4.x and G&D Smartcafe 6.0 (values 3 and 5 are avoided). Small factors in $p+1$ are not avoided by any source.

Concerning the *distribution of factors*, most of the software libraries (14) and card types (8) yield distributions comparable to that of randomly generated numbers of a given length (see Figure 13). The Infineon JTOP 80K card produces significantly more small factors than usual (compared with both random numbers and other sources). This decreases the probability of having a large factor.

We estimated the percentage of 512-bit RSA keys that are susceptible to Pollard $p-1$ factorization within 2^{80} operations. This percentage ranges from 0% (FIPS-compliant sources) to 4.35% (Infineon JCOP 80K), with an average of 3.38%. Although the NFS algorithm would still be faster in most cases of keys of 512 bits and larger, we found a card-generated key (with a small maximal factor of $p-1$) that was factorized via Pollard $p-1$ method in 19 minutes, whereas the NFS algorithm would require more than 2 000 CPU hours. Note that for 1024-bit keys, the probability of such a key being produced is negligible.

4.4 Sanity check

Based on the exported private and public components of the generated RSA keys obtained from all sources, we can summarize their basic properties as follows (see also Table 7):

- All values p and q are primes and are not close enough for Fermat factorization to be practical.
- All card-generated keys use a public exponent equal to `0x10001` (65 537), and all software libraries either use this value as the default or support a user-supplied exponent.
- Most modulus values are of an exactly required length (e.g., 1024 bits). The only exception is PGPSDK4 in FIPS mode, which also generates moduli that are shorter than the specified length by one or two bits.
- Neither libraries nor cards ensure that p is a safe prime ($p = 2 \cdot q + 1$, where q is also prime).
- Some sources construct strong primes according to the stricter definition or at least comply with the requirements defined in the FIPS 186-4 and ANSI X9.31 standards, such that $p - 1$ and $p + 1$ both have a large prime factor. Other libraries are not FIPS-compliant; however, keys of 1024 bits and larger resist $p - 1$ and $p + 1$ attacks for practical values of the smoothness bound.
- Some libraries (5) and most card types (12) order the primes such that $p > q$, which seems to be a convention for CRT RSA keys. PGPSDK4 (in both regular and FIPS modes) and libgcrypt (used by GnuPG) in both modes order the primes in the opposite manner, $q > p$. In some sources, the ordering is a side effect of the primes having fixed (and different) most significant bits (e.g., 4 bits of p and q are fixed to 1111 and 1001, respectively, by all G&D cards).
- All generated primes were unique for all libraries and all types of cards except one (Oberthur Cosmo Dual 72K).
- All G&D and NXP cards, the Oberthur Cosmo Dual 72K card and the GNU Crypto library generate Blum integers. As seen from a bug in the implementation of the Miller-Rabin test in GNU Crypto, a simpler version of the test suffices for testing Blum primes. However, we hypothesize that the card manufacturers have a different motivation for using such primes.

Source	Version	Classification group	Prime search method	Prime pair selection	Blum integers	Small factors of $p-1$	Large factor of $p-1$	Large factor of $p+1$	$\lvert p-q\rvert$ check	$\lvert d\rvert$ check	Notes
Open-source libraries											
Botan	1.11.29	XI	Incr.	11_2	×	✓	×	×	×	×	
Bouncy Castle	1.53	VIII	Incr.	RS	×	✓	×	×	✓	✓	Rejection sampling is less biased
Bouncy Castle	1.54	X	Incr.	$\sqrt{2}$	×	✓	×	×	✓	✓	Checks Hamming weight of the modulus
Cryptix JCE	20050328	VIII	Incr.	RS	×	✓	×	×	×	×	Rejection sampling is not biased
cryptlib	3.4.3	XI	Incr.	11_2	×	✓	×	×	✓	✓	
Crypto++	5.6.3	X	Incr.	$\sqrt{2}$	×	✓	×	×	×	×	$255 \geq$ MSB of prime $\geq 182 = \lceil\sqrt{2}\cdot 128\rceil$
FlexiProvider	1.7p7	VIII	Incr.	RS	×	✓	×	×	×	×	Rejection sampling is not biased
GNU Crypto	2.0.1	II	Rand.	RS	✓	✓	×	×	×	×	Rejection sampling is more biased
GPG Libgcrypt	1.6.5	XI	Incr.	11_2	×	✓	×	×	×	×	Used by GnuPG 2.0.30
GPG Libgcrypt	1.6.5 FIPS mode	XI	FIPS	11_2	×	✓	✓	✓	✓	×	101-bit prime factors of $p \pm 1$
LibTomCrypt	1.17	XI	Rand.	11_2	×	✓	×	×	×	×	
mbedTLS	2.2.1	VIII	Incr.	RS	×	✓	×	×	×	×	Rejection sampling is not biased
Nettle	3.2	XI	Maurer	11_2	×	✓	✓	×	×	×	Prime factor of $p-1$ has $(\lvert n\rvert/4+1)$ bits
OpenSSL	1.0.2g	V	Incr.	11_2	×	×	×	×	×	×	No prime factors 3 to 17 863 in $p-1$
OpenSSL FIPS	2.0.12	XI	FIPS	11_2	×	✓	✓	✓	✓	×	101-bit prime factors of $p \pm 1$
PGP SDK 4.x	PGP Desktop 10.0.1	XI	Incr.	11_2	×	✓	×	×	✓	×	p and q differ in their top 6 bits
PGP SDK 4.x	FIPS mode	IV	PGP	PGP	×	✓	✓	✓	✓	×	Prime factors of $p \pm 1$ have $(\lvert n\rvert/4-32)$ bits
SunRsaSign Provider	OpenJDK 1.8	VIII	Incr.	RS	×	✓	×	×	×	×	Rejection sampling is less biased
WolfSSL	3.9.0	XI	Rand.	11_2	×	✓	×	×	×	×	
Black-box implementations											
Microsoft CNG	Windows 10	X	FIPS	$\sqrt{2}$	×	✓	✓	✓	?	?	Prime factors of $p \pm 1$ have 101 to 120 bits
Microsoft CryptoAPI	Windows 10	X	FIPS	$\sqrt{2}$	×	✓	✓	✓	?	?	Prime factors of $p \pm 1$ have 101 to 120 bits
Microsoft .NET	Windows 10	X	FIPS	$\sqrt{2}$	×	✓	✓	✓	?	?	Prime factors of $p \pm 1$ have 101 to 120 bits

Table 7: *Comparison of cryptographic libraries. The algorithms are explained in Section 3.* ***Prime search method:*** *incremental search (Incr.); random sampling (Rand.); FIPS 186-4 Appendix B.3.6 or equivalent algorithm for strong primes (FIPS); Maurer's algorithm for provable primes (Maurer); PGP strong primes (PGP);* ***Prime pair selection:*** *practical square region (11_2); rejection sampling (RS); maximal square region ($\sqrt{2}$);* ***Blum integers:*** *the modulus n is always a Blum integer $n \equiv 1 \pmod 4$ (✓); the modulus is $n \equiv 1 \pmod 4$ and $n \equiv 3 \pmod 4$ with equal probability (×).* ***Small factors of*** $p-1$***:*** *$p-1$ contains small prime factors (✓); some prime factors are avoided in $p-1$ (×).* ***Large factors of*** $p-1$***:*** *$p-1$ is guaranteed to have a large prime factor – provable and strong primes (✓); size of the prime factors of $p-1$ is random (×).* ***Large factors of*** $p+1$***:*** *similar as for $p-1$, typically strong primes are (✓); random and provable primes are (×).* $\lvert p-q\rvert$ ***check:*** *p and q differ somewhere in their top bits (✓); the property is not guaranteed (×); the check may be performed, but the negative case occurs with a negligible probability (?).* $\lvert d\rvert$ ***check:*** *sufficient bit length of the private exponent d is guaranteed (✓); not guaranteed (×); possibly guaranteed, but not detectable (?).*

Source	Version	Classification group	Prime search method	Prime pair selection	Blum integers	Small factors of $p-1$	Large factor of $p-1$	Large factor of $p+1$	$\|p-q\|$ check	$\|d\|$ check	Notes
Smartcards											
Feitian JavaCOS A22		XI	Incr./Rand.	11_2	×	✓	×	×	?	?	
Feitian JavaCOS A40		XI	Incr./Rand.	11_2	×	✓	×	×	?	?	
G&D SmartCafe 3.2		XIII	Incr./Rand.	FX×9X	✓	✓	×	×	✓*	?	*Size of $\|p-q\|$ ensured by prime intervals
G&D SmartCafe 4.x		I	Incr./Rand.	FX×9X	✓	×	×	×	✓*	?	No prime factors 3 and 5 in $p-1$
G&D SmartCafe 6.0		I	Incr./Rand.	FX×9X	✓	×	×	×	✓*	?	No prime factors 3 and 5 in $p-1$
Gemalto GCX4 72K		XI	FIPS	11_2	×	✓	✓	✓	?	?	101-bit prime factors of $p \pm 1$
Gemalto GXP E64		IX	Gem.	Gem.	×	✓	✓	✓	?	?	112-bit prime factors of $p \pm 1$
Infineon JTOP 80K		XII	Inf.	Inf.	×	✓	×	×	?	?	
NXP J2A080		VII	Incr./Rand.	NXP	✓	✓	×	×	?	?	
NXP J2A081		VII	Incr./Rand.	NXP	✓	✓	×	×	?	?	
NXP J2D081		III	Incr./Rand.	RS	✓	×	×	×	✓	?	No prime factors 3 to 251 in $p-1$
NXP J2E145G		III	Incr./Rand.	RS	✓	×	×	×	✓	?	No prime factors 3 to 251 in $p-1$
NXP J3A081		VII	Incr./Rand.	NXP	✓	✓	×	×	?	?	
NXP JCOP 41 V2.2.1		VII	Incr./Rand.	NXP	✓	✓	×	×	?	?	
Oberthur Cosmo Dual 72K		VI	Incr.	11_2	✓	✓	×	×	?	?	
Oberthur Cosmo 64		XI	Incr./Rand.	11_2	×	✓	?	?	?	?	512-bit keys not supported

Table 8: *Comparison of smartcards. The algorithms are explained in Section 3.* ***Prime search method:*** *incremental search (Incr.); random sampling (Rand.); FIPS 186-4 Appendix B.3.6 or equivalent algorithm for strong primes (FIPS); Gemalto strong primes (Gem.); Infineon algorithm (Inf.); unknown prime generator with almost uniform distribution, possibly incremental or random search (Incr./Rand.).* ***Prime pair selection:*** *practical square region (11_2); rejection sampling (RS); maximal square region ($\sqrt{2}$); the primes p and q have a fixed pattern in their top four bits, 1111_2 and 1001_2, respectively (FX× 9X); Gemalto non-uniform strong primes (Gem.); Infineon algorithm (Inf.); NXP regions – 14 distinct square regions characterized by patterns in the top four bits of p and q (NXP).* ***Blum integers:*** *the modulus n is always a Blum integer $n \equiv 1 \pmod 4$ (✓); the modulus is $n \equiv 1 \pmod 4$ and $n \equiv 3 \pmod 4$ with equal probability (×).* ***Small factors of*** $p-1$***:*** *$p-1$ contains small prime factors (✓); some prime factors are avoided in $p-1$ (×).* ***Large factors of*** $p-1$***:*** *$p-1$ is guaranteed to have a large prime factor – provable and strong primes (✓); size of the prime factors of $p-1$ is random (×).* ***Large factors of*** $p+1$***:*** *similar as for $p-1$, typically strong primes are (✓); random and provable primes are (×).* $|p-q|$ ***check:*** *p and q differ somewhere in their top bits (✓); the property is not guaranteed (×); the check may be performed, but the negative case occurs with a negligible probability (?).* $|d|$ ***check:*** *sufficient bit length of the private exponent d is guaranteed (✓); not guaranteed (×); possibly guaranteed, but not detectable (?).*

5 Key source detection

The distinct distributions of specific bits of primes and moduli enable probabilistic estimation of the source library or card from which a given public RSA key was generated. Intuitively, classification works as follows: 1) Bits of moduli known to carry bias are identified with additional bits derived from the modulus value (a *mask*, $6 + 3$ bits in our method). 2) The frequencies of all possible *mask* combinations (2^9) for a given source in the learning set are computed. 3) For classification of an unknown public key, the bits selected by the *mask* are extracted as a particular value v. The source with the highest computed frequency of value v (step 2) is identified as the most probable source. When more keys from the same source are available (multiple values v_i), a higher classification accuracy can be achieved through element-wise multiplication of the probabilities of the individual keys.

We first describe the creation of a classification matrix and report the classification success rate as evaluated on our test set [55]. Later, classification is applied to three real-world datasets: the IPv4 HTTPS handshakes set [15], Certificate Transparency set [16] and the PGP key set [54].

5.1 The classification process

The classification process is not complicated and can be executed very quickly even for large sets of keys:

1. All modulus bits identified through previous analysis as non-uniform for at least one source are included in a *mask*. We included the $2^{\text{nd}} - 7^{\text{th}}$ most significant bits influenced by the prime manipulations described in Section 4.1, the second least significant bit (which is zero for sources that use Blum integers), the result of the modulus modulo 3 (which is influenced by the avoidance of factor 3) and the overall modulus length modulo 2 (which indicates whether an exact length is enforced).

2. A large number of keys (learning set) from known generating sources are used to create a *classification matrix*. For every possible *mask* value (of which there are 2^9 in our case) and every source, the relative frequency of the given mask value in the learning set for the given source is computed.

3. During the classification phase for key K with modulus m, the value v obtained after the application of *mask* to modulus m is extracted. The row (*probability vector*) of the *classification matrix* that corresponds to the value v contains, as its i^{th} element, the probability of K being produced by source i.

4. When a batch of multiple keys that are known to have been produced by the same (unknown) source is classified, the *probability vectors* for every key obtained in step 3 are multiplied element-wise and normalized to obtain the source probabilities p_b for the entire batch, and the source with the highest probability is selected.

Note that the described algorithm cannot distinguish between sources with very similar characteristics, e.g., between the NXP J2D081 and NXP J2E145G cards, which likely share the same implementation. For this reason, if two sources have the same or very similar profiles, they are placed in the same *group*. Figure 18 shows the clustering and (dis-)similarity of all sources considered in this study. If the particular source of one or more key(s) is missing from our analysis (relevant for the classification of real-world datasets), any such key will be misclassified as belonging to a group with a similar mask *probability vector*.

Both the construction of the classification matrix and the actual classification are then performed using these *groups* instead of the original single sources. The observed similarities split the examined sources into 13 different groups (labelled I to XIII and listed in Figure 18). The resulting classification matrix has dimensions of 13×512 and is available in Appendix B.

5.1.1 Evaluation of the classification accuracy

To evaluate the classification success of our method, we randomly selected 10 000 keys from the collected dataset (that were not used to construct the classification matrix) for every source, thereby endowing the test set with equal prior probability for every source.

A single organization may use the same source library to generate multiple keys for its web servers. The classification accuracy was therefore evaluated not only for one key (step 3 of the algorithm) but also for five, ten and one hundred keys (step 4) originating from the same (unknown) source. We evaluated not only the ability to achieve the "best match" with the correct source group but also the ability to identify the correct source group within the top two and top three most probable matches (top-n match).

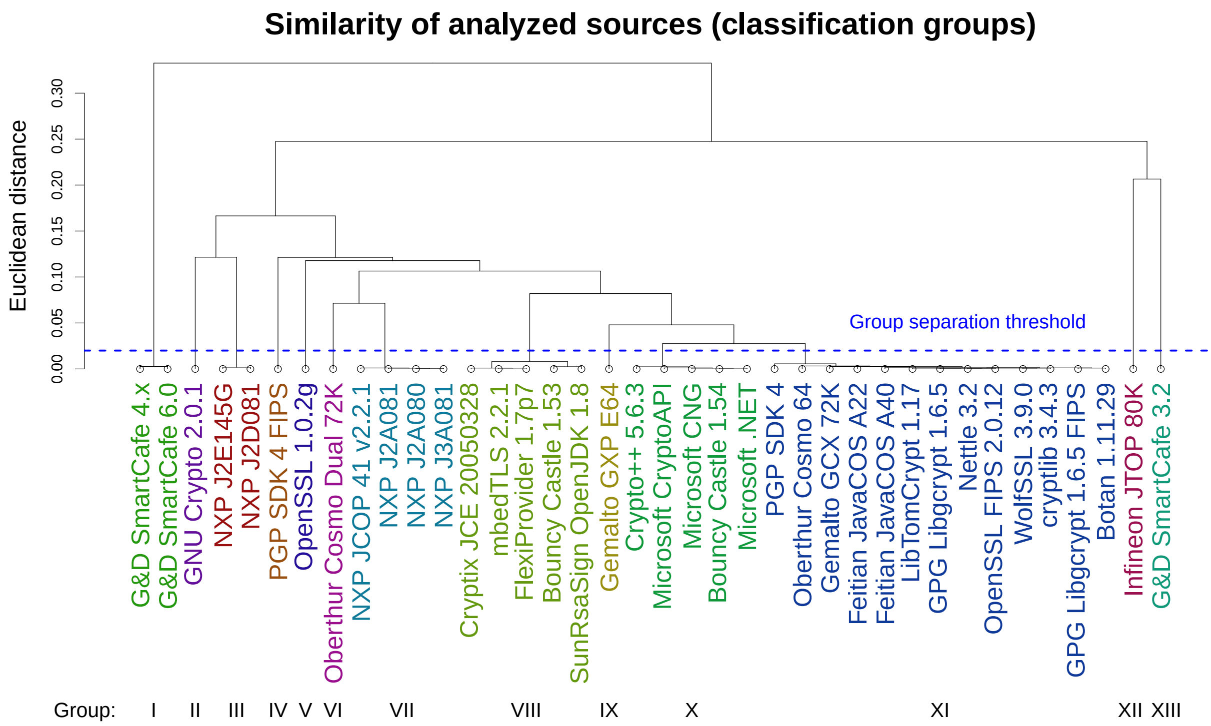


Figure 18: *Clustering of all inspected sources based on the 9 bits of the* mask. *The separation line shows which sources were put by us into the same classification category. Finer separation is still possible (e.g., SunRsaSign vs mbedTLS), but the number of the keys from same source needs to be high enough to distinguish these very similar sources.*

# of keys	Top 1 match					Top 2 match					Top 3 match				
in batch	1	2	5	10	100	1	2	5	10	100	1	2	5	10	100
Group I	95.39%	98.42%	99.38%	99.75%	100.00%	98.41%	99.57%	99.92%	100.00%	100.00%	98.41%	99.84%	100.00%	100.00%	100.00%
Group II	17.75%	32.50%	58.00%	69.50%	98.00%	35.58%	60.88%	84.15%	93.80%	100.00%	42.85%	71.58%	91.45%	98.40%	100.00%
Group III	45.36%	72.28%	93.17%	98.55%	100.00%	54.34%	78.31%	95.23%	99.35%	100.00%	82.45%	94.59%	99.25%	99.90%	100.00%
Group IV	90.14%	97.58%	99.80%	100.00%	100.00%	92.22%	98.14%	99.90%	100.00%	100.00%	94.42%	99.02%	100.00%	100.00%	100.00%
Group V	63.38%	81.04%	97.50%	99.60%	100.00%	84.14%	90.88%	99.25%	99.90%	100.00%	90.01%	96.62%	99.95%	100.00%	100.00%
Group VI	54.68%	69.22%	88.45%	94.60%	100.00%	80.31%	89.70%	97.90%	99.80%	100.00%	90.40%	96.34%	99.55%	100.00%	100.00%
Group VII	7.58%	31.69%	64.21%	82.35%	99.75%	32.67%	69.48%	95.33%	98.60%	100.00%	63.99%	88.70%	98.89%	99.70%	100.00%
Group VIII	15.65%	40.30%	68.46%	76.60%	85.20%	30.29%	52.81%	79.54%	92.38%	100.00%	39.32%	66.45%	90.34%	97.92%	100.00%
Group IX	22.22%	45.12%	76.35%	83.00%	83.00%	54.57%	71.86%	85.25%	86.80%	88.00%	61.77%	81.96%	94.35%	95.00%	99.00%
Group X	0.63%	6.33%	27.42%	42.74%	69.60%	15.05%	43.84%	78.83%	84.62%	91.00%	41.46%	70.54%	96.78%	99.88%	100.00%
Group XI	11.77%	28.40%	55.56%	65.28%	77.69%	29.94%	56.09%	86.43%	96.19%	100.00%	55.35%	78.48%	97.04%	99.77%	100.00%
Group XII	60.36%	79.56%	97.20%	99.40%	100.00%	82.96%	93.58%	99.60%	99.90%	100.00%	94.48%	97.62%	99.75%	100.00%	100.00%
Group XIII	39.56%	70.32%	96.20%	99.70%	100.00%	84.52%	95.54%	99.85%	100.00%	100.00%	95.22%	99.00%	99.95%	100.00%	100.00%
Average	**40.34%**	**57.90%**	**78.59%**	**85.47%**	**93.33%**	**59.62%**	**76.98%**	**92.40%**	**96.26%**	**98.38%**	**73.09%**	**87.75%**	**97.48%**	**99.27%**	**99.92%**

Table 9: *The classification success rate of 13 groups created from all 38 analyzed sources using test set with same prior probability of sources (see Figure 18 for libraries and cards in particular group). Columns corresponds to different number of keys (1, 2, 5, 10 and 100) classified together from same (unknown) source.*

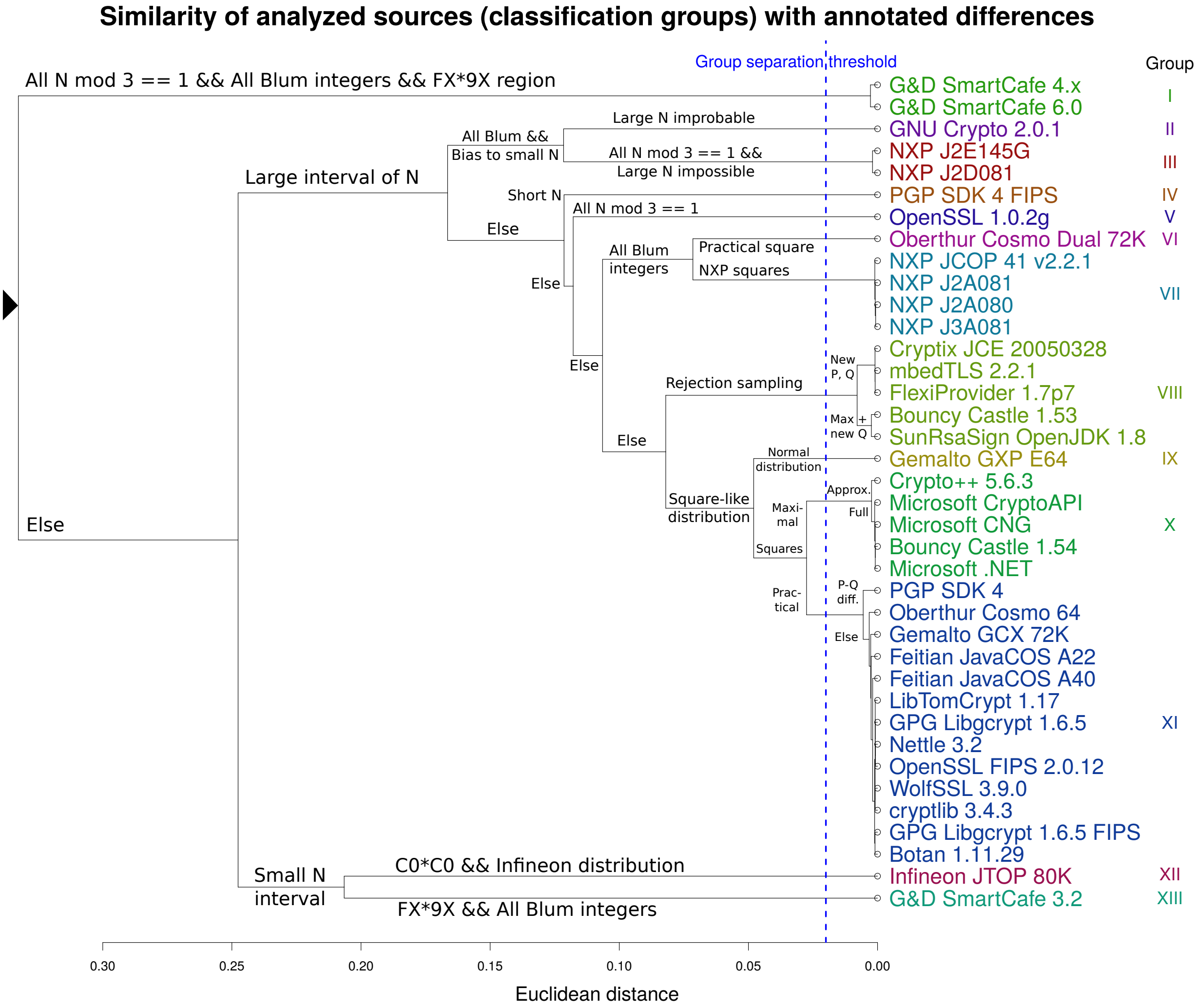
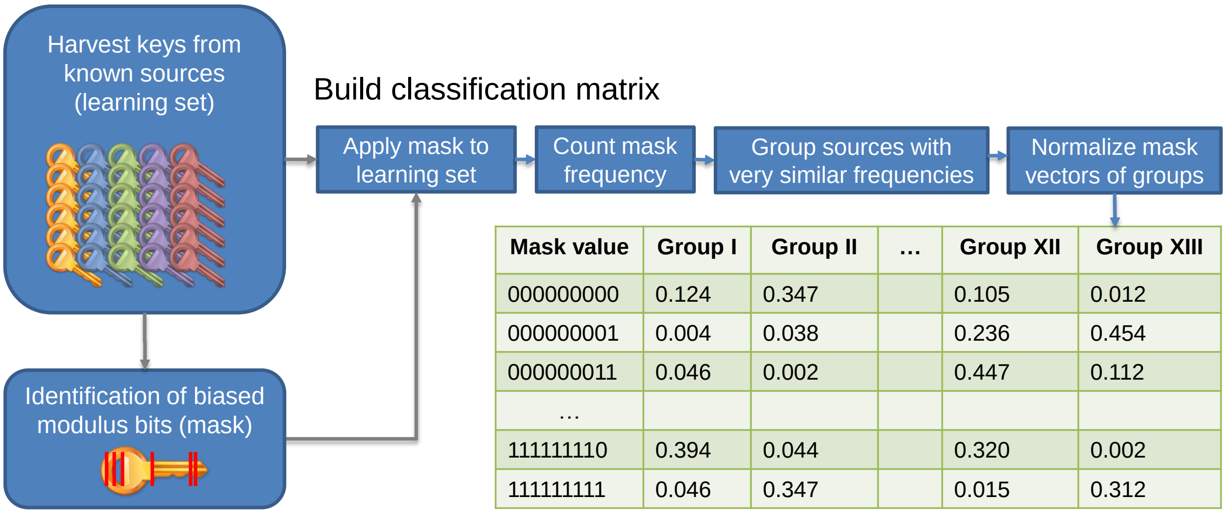


Figure 19: *Annotated version of Figure 18. The reasons for the clustering and the computed Euclidean distances of mask values can be traced back to differences in the implementations.*

As shown in Table 9, the average accuracy on the test set of the most probable source group was over 40% for single keys and improved to greater than 93% when we used batches of 100 keys from the same source for classification. When 10 keys from the same source were classified in a batch, the most probable classified group was correct in more than 85% of cases and was almost always (99%) included in the top three most probable sources.

A significant variability in classification success was observed among the different groups. Groups I (G&D cards) and IV (PGPSDK4 FIPS) could be correctly identified from even a single key because of their distinct distributions of possible mask values. By contrast, group X (Microsoft providers) was frequently misclassified when only a



Mask value	Group I	Group II	…	Group XII	Group XIII
000000000	0.124	0.347		0.105	0.012
000000001	0.004	0.038		0.236	0.454
000000011	0.046	0.002		0.447	0.112
…					
111111110	0.394	0.044		0.320	0.002
111111111	0.046	0.347		0.015	0.312

Figure 20: *The first phase of classification of the RSA public keys. The groups of similar sources are established based on the statistical properties of a large number of keypairs generated by separate sources. As a result, a fixed classification matrix is pre-computed.*

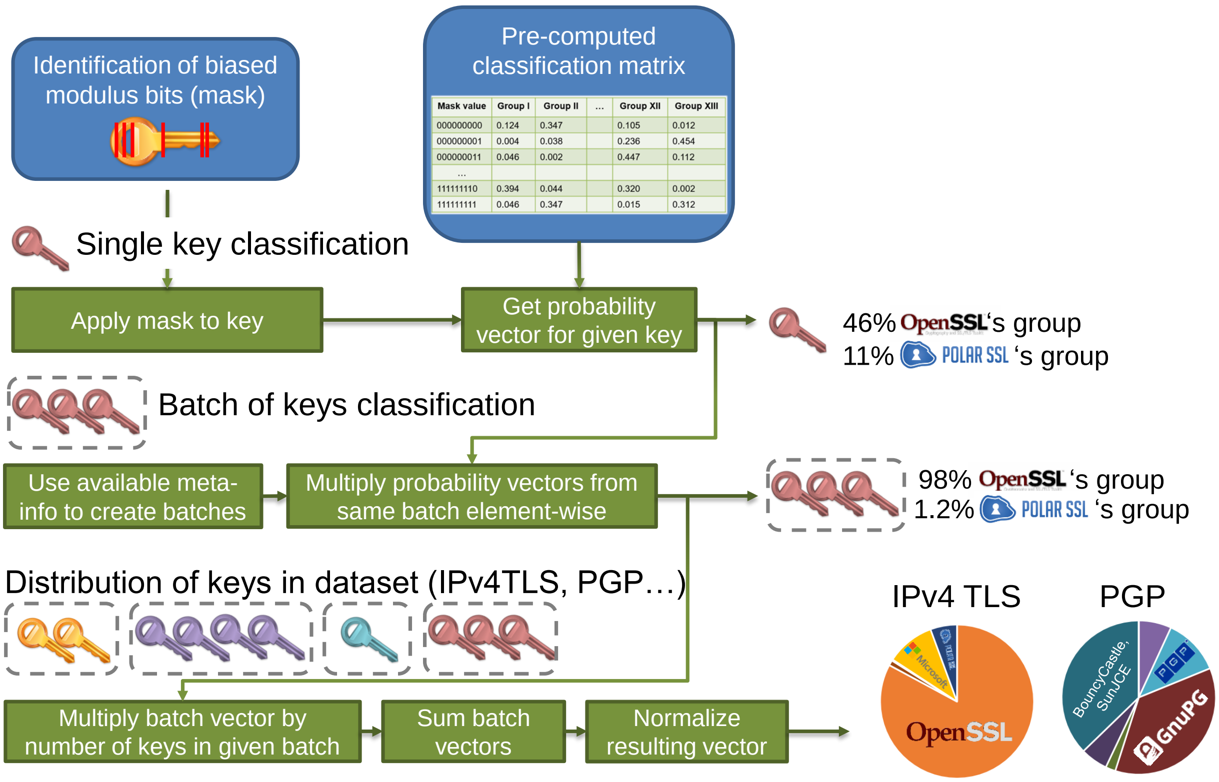


Figure 21: *An example of the second phase of classification. Either a) single key, b) batch of keys from the same source or c) whole dataset of batches is classified using pre-computed classification matrix.*

single key was used because of the wider range of possible mask values, resulting in a lower probability of each individual mask value.

We conclude that our classification method is moderately successful even for a single key and very accurate when a batch of at least 10 keys from the same source is classified simultaneously.

Further leakage in other bits of public moduli might be found by applying machine learning methods to the learning set, potentially leading to an improvement of the classification accuracy. Moreover, although we have already tested a wide range of software libraries and cards, more sources could also be incorporated, such as additional commercial libraries, various hardware security modules and additional types of cards and security tokens.

5.2 Classifying real-world keys

One can attempt to classify keys from suitable public datasets using the described method. However, the classification of keys observed in the real world may differ from the classification scenario evaluated above in two respects:

1. The prior probabilities of real-world sources can differ significantly (e.g., OpenSSL is a more probable source for TLS keys than is any card), and the resulting posterior probabilities from the classification matrix will then also be different.
2. Our classification matrix does not include all existing sources (e.g., we have not tested high-speed hardware security modules), and such sources will therefore always be misclassified.

The classification success rate can be significantly improved if the prior distribution of possible sources can be estimated. Such an estimate can be performed based on meta information such as statistics concerning the popularity of various software libraries or sales figures for a particular card model. Note that the prior distributions may also significantly differ for different application areas, e.g., PGP keys are generated by a narrower set of libraries and devices than are TLS keys. In this work, we did *not* perform any prior probability estimations.

5.2.1 Sources of Internet TLS keys

We used IPv4 HTTPS handshakes collected from the Internet-Wide Scan Data Repository [15] as our source of real-world TLS keys. The complete set contains approximately

Dataset (size of included batches)	#keys	Group of sources												
		I	II	III	IV	V	VI	VII	VIII	IX	X	XI	XII	XIII
Multiple keys classified in single batch, likely accurate results (see discussion in Section 5.1.1)														
TLS IPv4 (10-99 keys) [15]	518K	-	0.00%	-	0.01%	82.84%	-	-	1.09%	0.28%	10.18%	5.61%	-	-
TLS IPv4 (100+ keys) [15]	973K	-	-	-	0.01%	89.92%	-	-	4.68%	0.00%	3.46%	1.93%	-	-
Cert.Transparency (10-99 keys) [16]	23K	-	0.00%	-	0.07%	26.14%	-	-	6.90%	2.79%	47.70%	16.41%	-	-
PGP keyset (10-99 keys) [54]	1.7K	-	-	-	6.87%	11.95%	-	-	36.11%	2.09%	5.73%	37.25%	-	-
Classification based on batches with 2-9 keys only, likely lower accuracy results														
TLS IPv4 (2-9 keys) [15]	237K	0.02%	0.79%	2.06%	0.11%	54.14%	3.26%	1.73%	7.03%	7.98%	11.34%	11.17%	0.36%	0.05%
Cert. Transparency (2-9 keys) [16]	794K	0.03%	1.12%	3.21%	0.14%	43.89%	5.03%	2.64%	6.59%	10.52%	12.10%	14.18%	0.49%	0.06%
PGP keyset (2-9 keys) [54]	83K	0.02%	1.47%	1.40%	2.07%	14.36%	7.90%	3.91%	7.74%	16.10%	18.80%	25.86%	0.35%	0.03%
Classification based on single key only, likely low accuracy results														
TLS IPv4 (1 key) [15]	8.8M	0.98%	4.02%	6.47%	1.94%	21.01%	8.63%	6.13%	8.65%	12.22%	11.95%	13.48%	3.49%	1.03%
Cert. Transparency (1 key) [16]	12.7M	0.88%	3.75%	6.90%	1.49%	23.10%	8.69%	6.04%	7.99%	12.08%	11.78%	13.50%	3.04%	0.77%
PGP keyset (1 key) [54]	1.35M	0.44%	4.24%	4.09%	2.17%	13.91%	10.55%	7.18%	8.83%	14.34%	14.22%	16.79%	2.64%	0.59%

Table 10: *The ratio of resulting source groups identified by the classification method described in Section 5. Datasets are split into subsets based on the number of keys that can be attributed to a single source (batch). '-' means no key was classified for the target group. '0.00%' means that some keys were classified, but less than 0.005%.*

50 million handshakes; the relevant subset, which consists of handshakes using RSA keys with a public exponent of 65 537, contains 33.5M handshakes. This set reduces to 10.7M unique keys based on the modulus values. The keys in this set can be further divided into *batches* with the same subject and issue date (as extracted from their certificates), where the same underlying library is assumed to be responsible for the generation of all keys in a given *batch*. As the classification accuracy improves with the inclusion of more keys in a *batch*, we obtained classification results separately for batches consisting of a single key only (users with a single HTTPS server), 2-9 keys, 10-99 keys (users with a moderate number of servers) and 100 and more keys (users with a large number of servers).

Intuitively, batches with 100+ keys will yield very accurate classification results but will capture only the behaviour of users with a large number of HTTPS servers. Conversely, batches consisting of only a single key will result in low accuracy but can capture the behaviours of different types of users.

The frequency of a given source in a dataset (for a particular range of batch sizes) is computed as follows: 1) The classification probability vector p_b for a given batch is computed according to the algorithm from Section 5.1. 2) The element-wise sum of

$p_b \cdot n_b$ over all batches b (weighted by the actual number of keys n_b in the given batch) is computed and normalized to obtain the relative proportion vector, which can be found as a row in Table 10.

As shown in Section 5.1.1, a batch of 10 keys originating from the same source should provide an average classification accuracy of greater than 85% – sufficiently high to enable reasonable conclusions to be drawn regarding the observed distribution. Using batches of 10-99 keys, the highest proportion of keys generated for TLS IPv4 (82%) were classified as belonging to group V, which contains a single library – OpenSSL. This proportion increased to almost 90% for batches with 100+ keys. The second largest proportion of these keys (approximately 10.2%) was assigned to group X, which contains the Microsoft providers (CAPI, CNG, and .NET).

These estimates can be compared against the estimated distribution of commonly used web servers. Apache, Nginx, LiteSpeed, and Google servers with the OpenSSL library as the default option have a cumulative market share of 86% [56]. This value exhibits a remarkably close match to the classification rate obtained for OpenSSL (group V). MS Internet Information Services (IIS) is included with Microsoft's cryptographic providers (group X) and has a market share of approximately 12%. Again, a close match is observed with the classification value of 10.2% obtained for users with 10-99 certificates certified within the same day (batch).

Users with 100 and more keys certified within the same day show an even stronger preference for OpenSSL library (89.9%; group V) and also for group VIII (4.6%; this group contains popular libraries such as OpenJDK's SunRsaSign, Bouncy Castle and mbedTLS) at the expense of groups X and XI.

The classification accuracy for users with only single-key batches or a small number of keys per batch is significantly less certain, but the general trends observed for larger batches persist. Group V (OpenSSL) is most popular, with group X (Microsoft providers) being the second most common.

Another dataset of TLS keys was collected from Google's Pilot Certificate Transparency server [16]. The dataset processing was the same as that for the previous TLS dataset [15]. For users with small numbers of keys (1 and 2-9), the general trends observed from the TLS IPv4 dataset were preserved. Interestingly, however, Certificate Transparency dataset indicates that group X (Microsoft) is significantly more popular (47%) than group V (OpenSSL) for users with 10-99 keys.

Although we cannot obtain the exact proportions of keys in the the TLS handshake dataset generated using particular sources/groups, we can easily determine the proportion of keys that *certainly could not* have been generated by a given source by means of the occurrence of impossible values produced by the bit mask, i.e., values that are never produced by the given source. Using this method, we can conclude for certain that 19%, 25%, 17% and 10% of keys for users with 1, 2-9, 10-99 and 100+ keys per batch, respectively, could not have been generated by the OpenSSL library.

Apart from processing the TLS handshake dataset as a whole, we analyzed certificates issued by Let's Encrypt [18] certificate authority separately. While the classification results are not very precise due to small amount of keys that could be attributed to the same source, the negative classification gives us results with high confidence, especially for individual keys. At the time of the analysis, the recommended tool for Let's Encrypt certificate requests was using the OpenSSL library. It can be seen from Table 11 that only 1.83% of keys used in individual certificates issued by Let's Encrypt are definitely not produced by OpenSSL, while the other datasets contain more keys certainly coming from other libraries.

5.2.2 Sources of PGP keys

A different set of real-world keys can be obtained from PGP key servers [54]. We used a dump containing nearly 4.2 million keys, of which approximately 1.4 million were RSA keys suitable for classification using the same processing as for the TLS datasets. In contrast to the TLS handshakes, significantly fewer PGP keys could be attributed to the same batch (i.e., could be identified as originating from the same unknown source) based on the subject name and certification date. Still, 84 thousand unique keys were extracted in batches of 2-9 keys and 1 732 for batches of 10-99 keys.

The most prolific source group is group XI (which contains both libgcrypt from the GnuPG software distribution and the PGPSDK4 library), as seen in Table 10. This is intuitively expected because of the widespread use of these two software libraries. Group VIII, consisting of the Bouncy Castle library (containing the *org.bouncycastle.openpgp* package), is also very common (36%) for batches of 10-99 keys.

Because of the lower accuracy of classification for users with smaller numbers of keys (1 and 2-9), it is feasible only to consider the general properties of these key batches and their comparison with the TLS case rather than being concerned with the exact percentage values in these settings. The results for the PGP dataset indicate a significant

Dataset	Group of sources												
	I	II	III	IV	V	VI	VII	VIII	IX	X	XI	XII	XIII
Individual keys – the percentage of keys that were not generated by any source from given group – the results are certain													
Cert. Transparency [16]	96.97%	48.61%	57.29%	0.00%	11.80%	50.69%	51.55%	0.00%	1.78%	0.00%	1.40%	83.99%	96.02%
PGP keyset [54]	98.46%	49.22%	74.49%	0.00%	47.35%	50.14%	51.55%	0.42%	2.25%	0.42%	0.76%	86.04%	97.11%
TLS IPv4 [15]	96.69%	48.77%	60.38%	0.00%	18.91%	51.44%	51.73%	0.39%	2.56%	0.39%	2.90%	82.67%	95.14%
Let's Encrypt [15]	97.29%	48.54%	52.74%	0.00%	1.83%	49.88%	51.43%	0.00%	1.34%	0.00%	0.02%	86.05%	97.23%
Batch of 2-9 keys from the same source – the percentage of keys that were not generated by any source from given group – possibly imprecise due to batch creation													
Cert. Transparency [16]	99.95%	79.98%	86.65%	0.00%	24.57%	81.52%	82.35%	0.01%	4.59%	0.01%	2.87%	98.10%	99.84%
PGP keyset [54]	99.95%	77.68%	93.97%	0.00%	71.99%	78.52%	79.80%	1.83%	6.18%	1.83%	2.68%	98.38%	99.90%
TLS IPv4 [15]	99.96%	87.18%	91.88%	0.00%	25.54%	88.44%	88.88%	0.02%	7.16%	0.02%	5.58%	98.81%	99.88%
Let's Encrypt [15]	100.00%	69.81%	73.58%	0.00%	5.03%	69.81%	71.07%	0.00%	3.77%	0.00%	0.00%	97.48%	100.00%
Batch of 10-99 keys from the same source – the percentage of keys that were not generated by any source from given group – possibly imprecise due to batch creation													
Cert. Transparency [16]	100.00%	99.95%	100.00%	0.00%	73.85%	100.00%	100.00%	0.07%	35.90%	0.07%	40.22%	100.00%	100.00%
PGP keyset [54]	100.00%	100.00%	100.00%	0.00%	88.05%	100.00%	100.00%	6.87%	59.76%	6.87%	43.24%	100.00%	100.00%
TLS IPv4 [15]	100.00%	99.99%	100.00%	0.00%	17.15%	100.00%	100.00%	0.00%	43.82%	0.00%	9.12%	100.00%	100.00%
Let's Encrypt [15]	-	-	-	-	-	-	-	-	-	-	-	-	-
Batch of 100+ keys from the same source – the percentage of keys that were not generated by any source from given group – possibly imprecise due to batch creation													
Cert. Transparency [16]	100.00%	100.00%	100.00%	0.00%	1.53%	100.00%	100.00%	0.00%	99.64%	0.00%	0.82%	100.00%	100.00%
PGP keyset [54]	100.00%	100.00%	100.00%	0.00%	100.00%	100.00%	100.00%	0.00%	100.00%	0.00%	0.00%	100.00%	100.00%
TLS IPv4 [15]	100.00%	100.00%	100.00%	0.00%	10.08%	100.00%	100.00%	0.01%	96.90%	0.01%	8.12%	100.00%	100.00%
Let's Encrypt [15]	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 11: *The percentage of keys that were not generated by any source from given group. While the classification results for single keys are the least precise, the negative results are obtained with certainty for individual keys. The precision of the claim for batches of keys depends only on the correct creation of batches coming from the same source (determined by accompanying meta-data). The group IV (PGPSDK 4 FIPS) could never be ruled out since it's the only source which generates keys with all possible mask values. On the contrary, sources which generate keys with only a limited range of mask values are ruled out for most of the keys (groups I, XII and XIII). Groups VIII and X generate keys with half of the possible mask values, however, they are almost never ruled out. The missing mask values are characterized by having moduli of odd length, which shows that most keys had even modulus length (512, 1024, etc.), as we were able to confirm from the datasets. The group V (OpenSSL) generates keys such that the modulus is always congruent to 1 modulo 3, hence producing only half of possible mask values for the even length of keys. When a set of keys is congruent to 1 and 2 modulo 3 with equal probability, approximately half of the keys will be marked as possibly coming from OpenSSL and half will be impossible to be generated by OpenSSL. Such percentage can be observed for single keys in the PGP dataset, where OpenSSL will be very rarely used since it does not normally support PGP. On the contrary, the keys used with the Let's Encrypt certificate authority are very rarely ruled out as coming from OpenSSL. The result is expected, since the default tools for generating Let's Encrypt certificate requests use OpenSSL and only servers using customized tools will generate keys with other libraries. The OpenSSL percentages for the TLS and Certificate Transparency datasets also diverge from 50%, influenced by the a priori probabilities of OpenSSL usage. The statistics for Let's Encrypt are not available for larger batches (the symbol '-') since there was no source that could be reliably attributed with 10 or more keys in the dataset. The services of the Let's Encrypt certificate authority may be primarily used by small web service providers.*

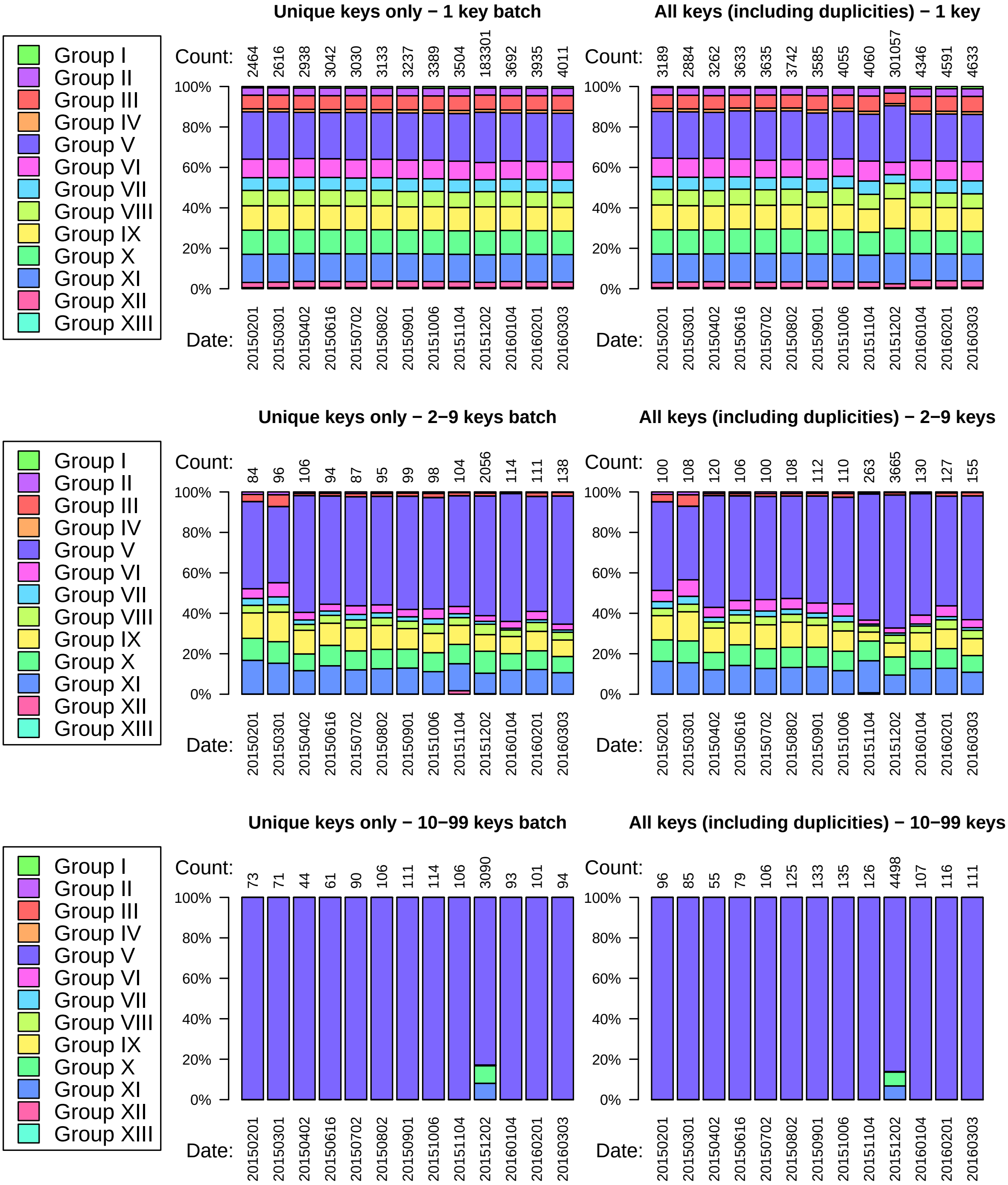


Figure 22: *The change in the distribution of keys in the TLS dataset over time. Each bar represents the percentage of libraries used to generate keys with validity starting no sooner than the corresponding label but no later than the following bar label. Datasets are spaced approximately one month apart. Significantly more keys have validity starting between 02 December 2015 and 04 January 2016 than in other months. Some keys appear in more than one certificate, causing the difference between unique key classification and results for the dataset with duplicities.*

drop in the proportion of keys generated using the OpenSSL library. According to an analysis of the keys that *certainly could not* have been obtained from a given source, at least 47% of the single-key batches were certainly *not* generated by OpenSSL, and this percentage increases to 72% for batches of 2-9 keys. PGPSDK4 in FIPS mode (group IV) was found to be significantly more common than in the TLS datasets.

Note that an exported public PGP key usually contains a *Version* string that identifies the software used. Unfortunately, however, this might be not the software used to generate the original key pair but merely the software that was used to export the public key. If the public key was obtained via a PGP keyserver (as was the case for our dataset), then the *Version* string indicates the version of the keyserver software itself (e.g., *Version: SKS 1.1.5*) and cannot be used to identify the ratios of the different libraries used to generate the keys[9].

5.3 Practical impact of origin detection

The possibility of accurately identifying the originating library or card for an RSA key is not solely of theoretical or statistical interest. If some library or card is found to produce weak keys, then an attacker can quickly scan for other keys from the same vulnerable source. The possibility of detection is especially helpful when a successful attack against a weak key requires a large but practically achievable amount of computational resources. Preselecting potentially vulnerable keys saves an attacker from spending resources on all public keys.

The identification of the implementations responsible for the weak keys found in [4, 20] was a difficult problem. In such cases, origin classification can quickly provide one or a few of the most probable sources for further manual inspection. Additionally, a set of already identified weak keys can be used to construct a new classification group, which either will match an already known one (for which the underlying sources are known) or can be used to search for other keys that belong to this new group in the remainder of a larger dataset (even when the source is unknown).

Another practical impact is the decreased anonymity set of the users of a service that utilizes the RSA algorithm whose users are not intended to be distinguishable (such as the Tor network). Using different sources of generated keys will separate users into smaller anonymity groups, effectively decreasing their anonymity sets. The resulting

[9] A dataset with the original *Version* strings could be used to test these predictions.

anonymity sets will be especially small when individual users decides to use cryptographic hardware to generate and protect their private keys (if selected device does not fall into into group with widely used libraries). Note that most users of the Tor project use the default client, and hence the same implementation, for the generation of the keys they use. However, the preservation of indistinguishability should be considered in the development of future alternative clients.

Tor hidden services sometimes utilize ordinary HTTPS certificates for TLS [1], which can be then linked (via classification of their public keys) with other services of the same (unknown) operator.

Mixnets such as mixmaster and mixminion use RSA public keys to encrypt messages for target recipient and/or intermediate mix. If key ID is preserved, one may try to obtain corresponding public key from PGP keyserver and search for keys with the same source to narrow that user's anonymity set in addition to analysis like one already performed on alt.anonymous.messages [43]. Same as for Tor network, multiple seemingly independent mixes can be linked together if uncommon source is used to generate their's RSA keys.

A related use is in a forensic investigation in which a public key needs to be matched to a suspect key-generating application. Again, secure hardware will more strongly fingerprint its user because of its relative rarity.

An interesting use is to verify the claims of remote providers of Cryptography as a Service [6] regarding whether a particular secure hardware is used as claimed. As the secure hardware (cards) used in our analysis mostly exhibit distinct properties of their generated keys, the use of such hardware can be distinguished from the use of a common software library such as OpenSSL.

5.4 How to mitigate origin classification

The impact of successful classification can be mitigated on two fronts: by library maintainers and by library users. The root cause lies with the different design and implementation choices for key generation that influence the statistical distributions of the resulting public keys. A maintainer can modify the code of a library to eliminate differences with respect to the approach used by all other sources (or at least the most common one, which is OpenSSL in most cases). However, although this might work for one specific library (mimicking OpenSSL), it is not likely to be effective on a wider scale. Changes to all major libraries by its maintainers are unlikely to occur, and many users will continue to use older versions of libraries for legacy reasons.

More pragmatic and immediate mitigation can be achieved by the users of these libraries. A user may repeatedly generate candidate key pairs from his or her library or device of choice and reject it if its classification is too successful. Expected number of trials differs based on the library used and the prior probability of sources within the targeted domain. For example, if TLS is the targeted domain, five or less key generation trials are expected for most libraries to produce "indecisive" key.

The weakness of the second approach lies in the unknown extent of public modulus leakage. Although we have described seven different causes of leakage, others might yet remain unknown – allowing for potential future classification of keys even after they have been optimized for maximal indecisiveness against these seven known causes.

This strategy can be extended when more keys are to be generated. All previously generated keys should be included in a trial classification together with the new candidate key. The selection process should also be randomized to some extent; otherwise, a new classification group of "suspiciously indecisive" keys might be formed.

6 Random numbers generated on cards

Truly random data generated on-card are a crucial input for the primes used in RSA key pair generation. A bias in these data would influence the predictability of the primes. If a highly biased or malfunctioning generator is used, factorization is not necessary (only a small number of fixed values can be taken as primes) or is feasible even for RSA keys with lengths otherwise deemed to be secure [4, 10, 20].

6.1 Biased random number generator

The output of an on-card truly random number generator (TRNG) can be tested using statistical batteries, and deviances are occasionally detected in commercial security tokens [10]. We generated a 100 MB stream of random data from one card of each type and tested these data streams using the common default settings of the NIST STS and the Dieharder battery of statistical tests [11, 46] as well as our alternative EACirc distinguisher [48]. All types of cards except two (Infineon JTOP 80K and Oberthur Cosmo Dual 72K) passed the tests with the expected number of failures at a confidence level of 1%.

The Infineon JTOP 80K failed the NIST STS Approximate Entropy test (85/100, expected entropy contained in the data) at a significant level and also failed the group of Serial tests from the Dieharder suite (39/100, frequency of overlapping n-bit patterns). Interestingly, the serial tests began to fail only for patterns with lengths of 9 bits and longer (lengths of up to 16 bits were tested), suggesting a correlation between two consecutive random bytes generated by the TRNG. As shown in Figure 23, for 16-bit patterns in the overlapping serial test, all bytes in the form of $xyxy$ (where x and y denote 4-bit values) were 37% less likely to occur than other combinations. At least three more distinct groups of inputs with smaller-than-average probabilities were also identified. Note that deviating distributions were observed in all three physical Infineon JTOP 80K cards that were tested and thus were probably caused by a systematic defect in the entire family of cards rather than a single malfunctioning device. The detected bias is probably not sufficient to enable faster factorization by guessing potential primes according to the slightly biased distribution. However, it may be used to identify this type of card as the source of a sufficiently large (e.g., 1KB) random data stream (i.e., to fingerprint such a random stream).

The Oberthur Cosmo Dual 72K failed more visibly, as two cards were blocked after the generation of only several MB of random data. The statistical tests then frequently failed because of the significant bias in the data. Several specific byte values were never produced in the "random" stream. See Figure 24 for more details and Figure 25 for expected frequencies of non-overlapping serial tests.

We also generated data streams directly from the concatenated exported primes with the two most significant bytes and the least two bits dropped, as the previous analysis had revealed a non-uniform distribution in these bits. Interestingly, both the Infineon JTOP 80K and the Oberthur Cosmo Dual 72K failed only for their random data streams (as described above) but successfully passed[10] for the streams generated from the concatenated primes, hinting at the possibility that either random data are generated differently during prime generation or (unlikely) the prime selection process is able to mask the bias observed in the raw random data.

[10] Except for the Oberthur nearly zero keys (see Section 7.1).

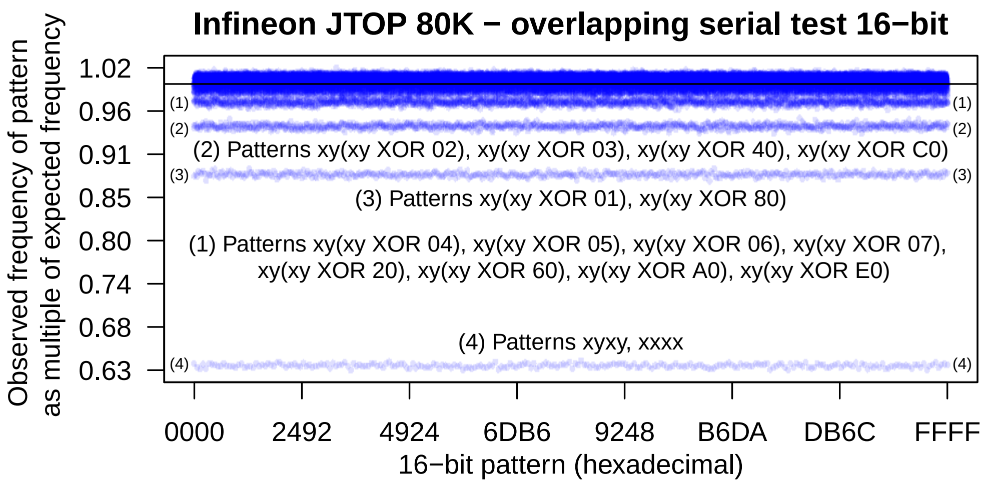


Figure 23: *The frequencies of different patterns with the length of 16 bits computed from 1 GB random data stream generated by the Infineon JTOP 80K card. The floating window with length of 16 bits moved by one bit every time is used (overlapping serial test). At least five distinct patterns can be identified where patterns should exhibit uniform distribution instead. Note that overlapping frequency calculation provides the consistent result no matter which particular starting bit is used (e.g., if original generator will not output generated bits aligned to multiple of whole bytes), but may make identification of real defect more difficult. As can seen on Figure 25 where non-overlapped frequencies are shown, five visibly distinct patterns are all caused by single property of the generated stream – two subsequent bytes will never have the same value.*

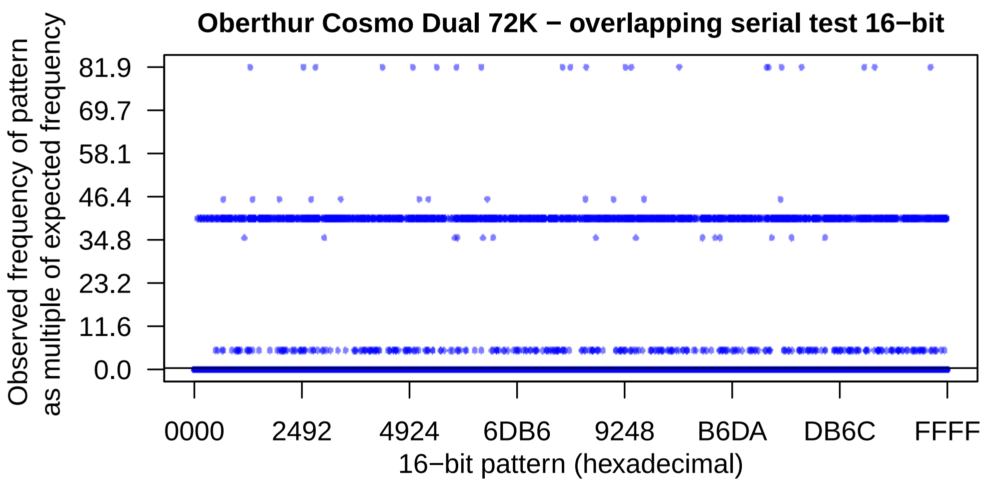


Figure 24: *The frequencies of different patterns with the length of 16 bits computed from 2 MB random data stream generated by the Oberthur Cosmo Dual 72K card. Multiple distinct groups of patterns can be identified with some patterns never occurring in the tested random stream where patterns should be uniformly distributed instead.*

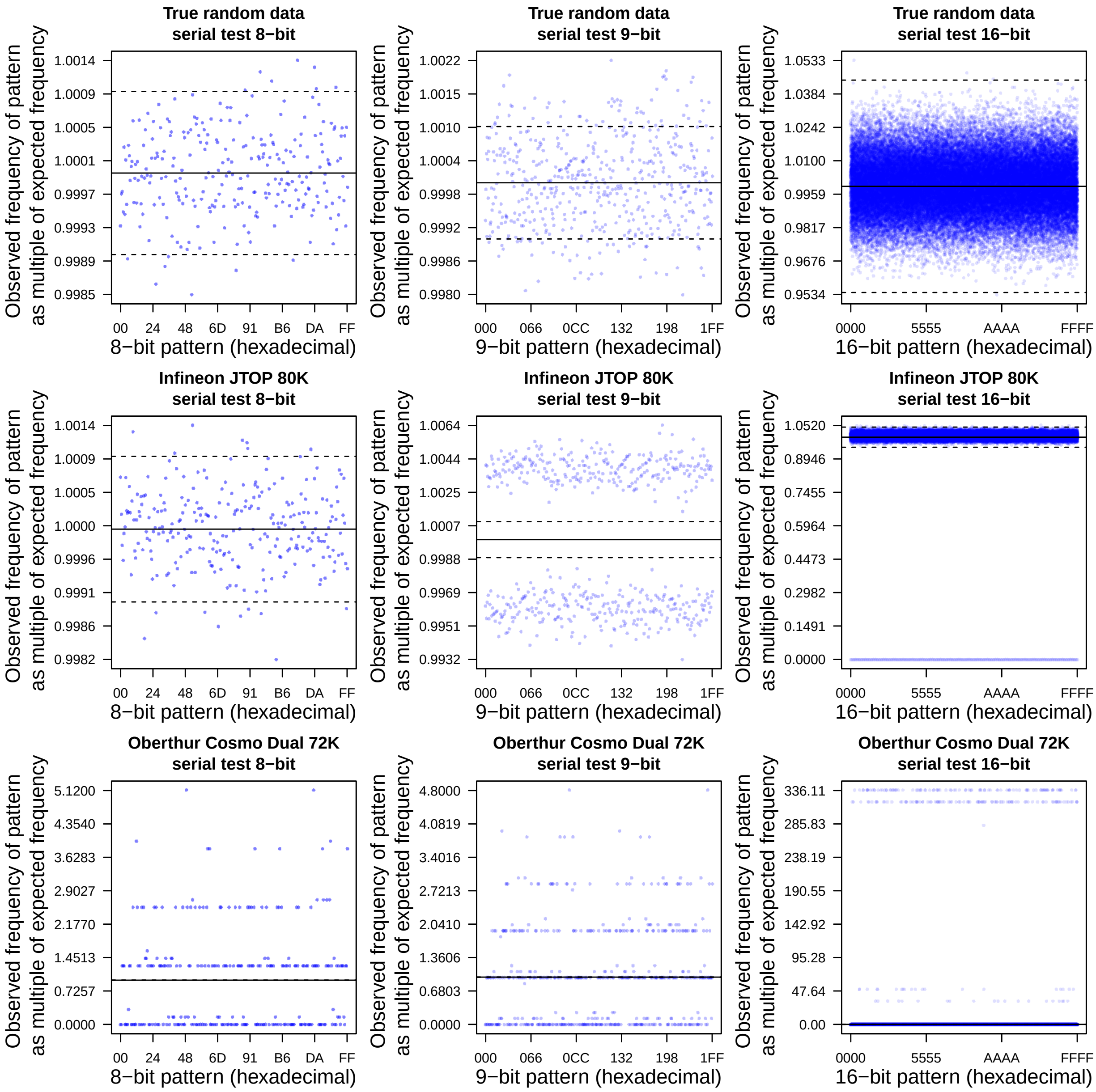


Figure 25: *Comparison of the expected and observed frequencies of different patterns with the length of 8, 9 and 16 bits for non-overlapping serial test. The frequencies are computed from 1 GB random data stream generated by a truly random generator [35] (expected distribution) and Infineon JTOP 80K card. The results for Oberthur Cosmo Dual 72K are generated for 2 MB of random data, since the card became unresponsive after generating just a small amount of data. Two subsequent bytes output by Infineon JTOP 80K generator are never identical. The occurrence of such pair is expected for a truly random number generator. The data generated from Oberthur Cosmo Dual 72K exhibit more significant bias. In the stream of 1 919 400 bytes, out of 256 possible 8-bit values, only 173 occurred (while just 154 occurring more than twice). Out of 65 536 possible pairs of subsequent bytes only 254 appeared (235 more than once).*

7 Key generation process on cards

The algorithms used in open-source libraries can be inspected and directly correlated to the biases detected in their outputs. To similarly attribute the biased keys produced by cards to their unknown underlying algorithms, we first verified whether the random number generator might instead be responsible for the observed bias in Section 6. We also examined the time- and power-consumption side channels of the cards to gain insight into the processes responsible for key generation.

7.1 Malfunctioning generator

All primes for the card-generated 512- and 1024-bit keys were tested for uniqueness. All tested card types except one generated unique primes. In the exceptional case of the Oberthur Cosmo Dual 72K cards, approximately 0.05% of the generated keys shared a specific value of prime `q`. The flaw was discovered in all three tested physical cards for both 512-bit and 1024-bit keys. The repeated prime value was equal to `0xC000...0077` for 512-bit RSA keys and `0xC000...0E9B` for 1024-bit RSA keys. These prime values correspond to the first Blum prime generated when starting from the value `0xC000...0000` in each case.

The probable cause of such an error is the following sequence of events during prime generation: 1) The random number generator of the card was called but failed to produce a random number, either by returning a value with all bits set to zero or by returning nothing into the output memory, which had previously been zeroed. 2) The candidate prime value `q` (equal to 0 at the time) had its highest four bits fixed to 1100_2 (to obtain a modulus of the required length[11] when multiplied by the prime `p`), resulting in a value of `0xC0` in the most significant byte. 3) The candidate prime value was tested for primality and increased until the first prime with the required properties (a Blum prime in the case of the Oberthur Cosmo Dual 72K) was found (`0xC000...0077` in the case of 512-bit RSA).

The faulty process described above that leads to the observed predictable primes may also occur for other cards or software libraries as a result of multiple causes (e.g., an ignored exception in random number generation or a programming error). We therefore inspected our key pair dataset, the TLS IPv4 dataset [15] and the PGP dataset [54]

[11] As was observed for the dataset analysed in Section 4.

for the appearance of such primes relevant to key lengths of 512, 1024 and 2048 bits. Interestingly, no such corrupt keys were detected except for those already described.

Note that a random search for a prime is much less likely to fail in this mode when compared to incremental search. Even if some of the top bits and the lowest bit are set to one, the resulting value is not a prime for common MSB masks. New values will be generated if the starting value contains only zeroes.

7.2 Time distribution

We measured the time of the key generation process[12]. The length of the process differs, affected by the time it takes to find two primes. We expect that other parts of the process, such as computation of private key, take almost the same time in every run. We experimentally obtained the distribution for amount of random odd numbers it takes to find a prime by random search. The distribution corresponds also to the distance from random point to nearest prime, as illustrated by Figure 26. Distribution of sum of distances for `p` and `q` creates a log-normal distribution, similar as we typically observe for the time of the process in a software library.

We experimentally obtained distributions for a number of needed primality tests for different parameters of trial division (Figure 27). Then we were able to match them with distributions from several cards, obtaining a likely estimate for the number of primes used by the card in the trial division (sieving) phase. For some types of cards, a single parameter did not match distributions of neither 512-bit nor 1024-bit keys. There may exist a different optimal value of trial division tests and primality tests for different key lengths. Notably, in some cases of card-generated 512-bit keys, the number of primality tests would have to be halved to exactly match a referential distribution. However, we are not aware of a mechanism that would perform two primality tests in parallel or at least in the same time, as is required for testing a candidate of double bit length.

The exact time distribution for software implementations is of less concern since the key generation process tends to be much faster on an ordinary CPU. The source code can be modified to accommodate for counting the number of tests directly (as shown in the inlay in Figure 28) without relying on time measurement that may be influenced by other factors specific to the implementation.

[12]Usually measured in number of clock ticks. The time does not include the period when the system gathers entropy for `/dev/random` and other interrupts, which is the longest part of the process in some libraries, such as libgcrypt. The key generation time on cards was measured in milliseconds.

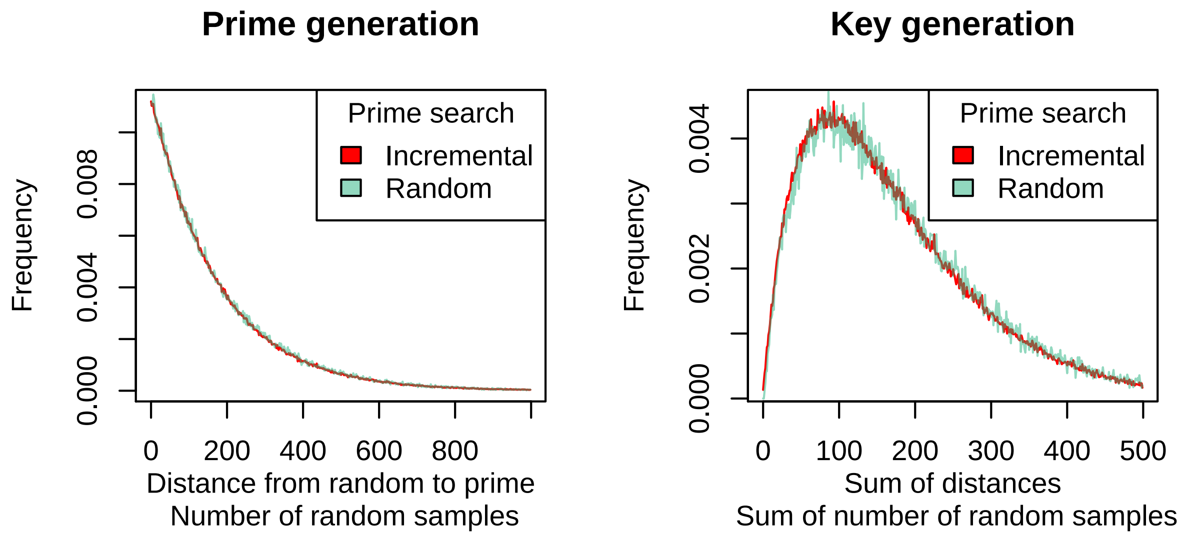


Figure 26: *Left: distribution of amount of random odd numbers generated, until a prime is found, exactly matches the distribution of distance from random number until a prime is found by incremental search (in 2-increments). Right: summing the number of trials for* p *and* q *creates a log-normal distribution.*

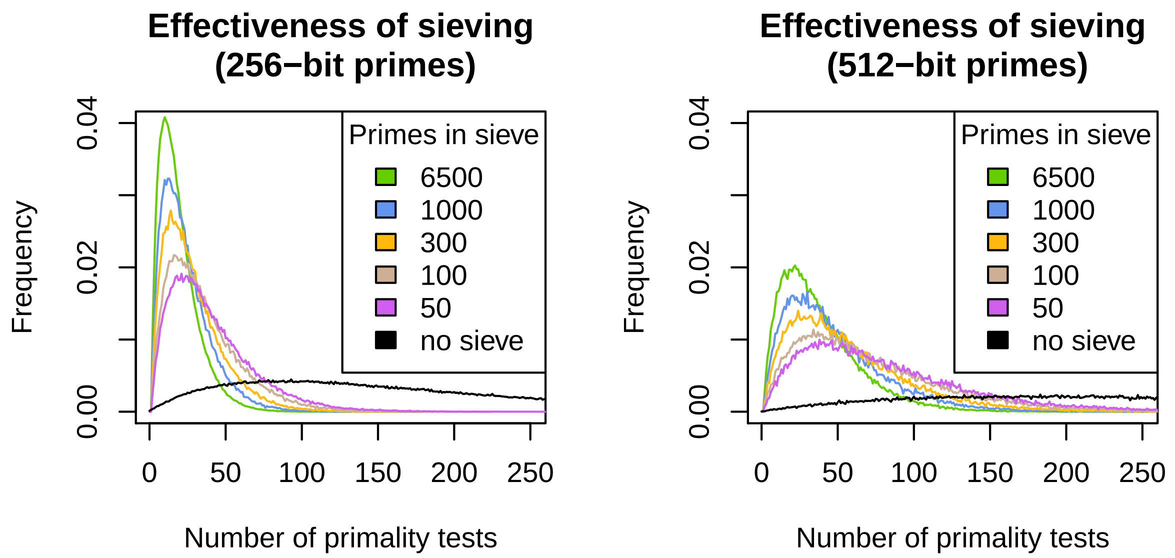


Figure 27: *Trial division (or sieving) by a different number of small primes. Testing a candidate with division by a few primes decreases the number of required probabilistic primality tests and speeds up prime generation. The expected number of primality tests decreases when sieving phase runs with more primes.*

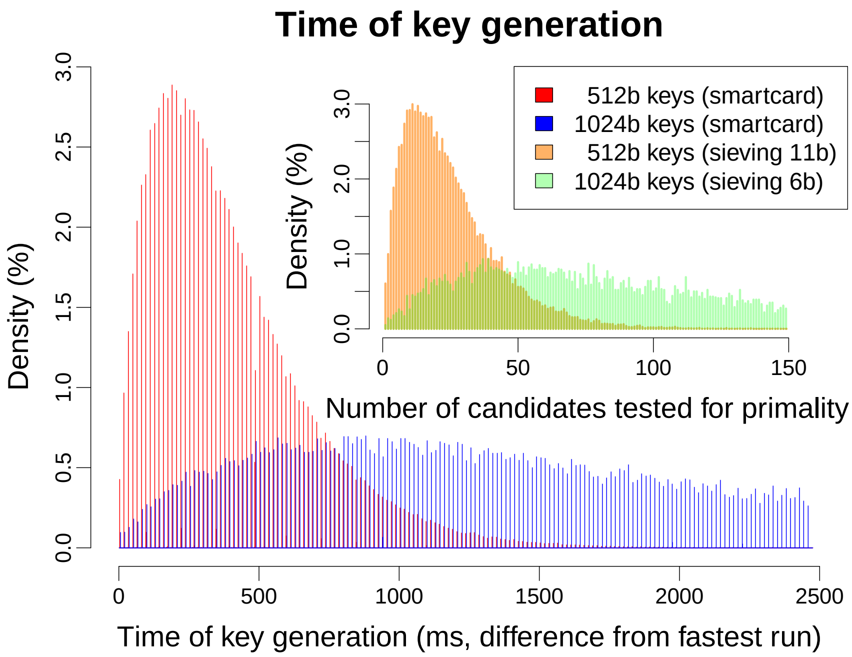


Figure 28: *An example of the histogram of times necessary to generate a large number of 512 and 1024-bit RSA keys generated from an NXP J2D081 card. Left – the distribution of key generation times is concentrated around evenly spaced points, with the distance representing the duration of a single primality test. The times were normalized to begin at zero, therefore they represent difference from the fastest run. Inlay – the distribution of number of candidates tested by primality tests obtained from a software implementation. 512-bit keys are generated with trial division up to 11-bit primes, 1024-bit keys used 6-bit primes. The results show a clear correlation between the generation time and an expected number of primality tests.*

7.3 Power analysis

Analysis of power consumption traces is a frequently used technique for card inspection. The baseline power trace expected should cover at least the generation of random numbers of potential primes, primality testing, computation of the private exponent and storage of generated values into a persistent key pair object. We utilized the simple power analysis to reveal significant features like random number generation, RSA encryption, and RSA decryption operation, separately. By programming a card to call only the desired operation (generate random data, encrypt, decrypt), the feature pattern for the given operation is obtained. These basic operations were identified in all tested types of cards. Once identified, the operations can be searched for inside a more

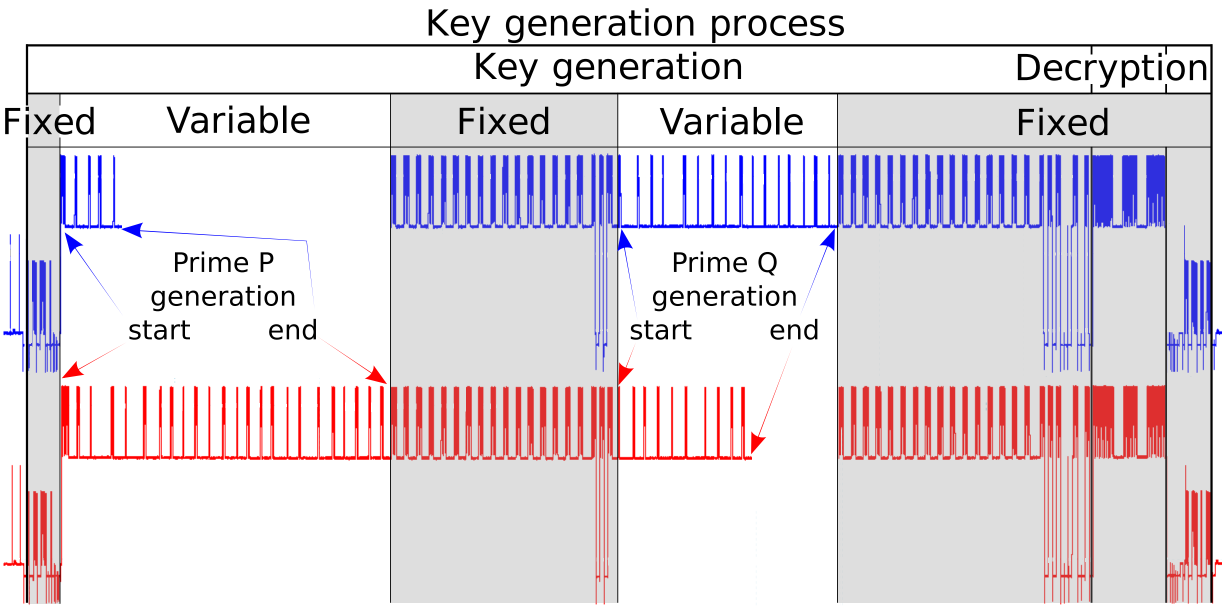


Figure 29: *Annotated example power traces showing typical steps of on-card key pair generation. The upper trace is shorter in the first variable part (potentially corresponding to generation of first prime and longer in the second variable part). Both traces are intentionally interrupted to visually synchronize at fixed parts.*

complex operations like the RSA key pair generation. Please refer to [27] for images of collected and annotated power traces from all inspected cards.

A typical trace of the RSA key pair generation process (although feature patterns may differ with card hardware) contains: 1) Power consumption increases after the generating key pair method is called (cryptographic RSA co-processor turned on). 2) Candidate values for primes p and q are generated (usage of a TRNG can be observed from the power trace) and tested. 3) The modulus and the private exponent are generated (assumed, not distinguishable from the power trace). 4) Operation with a private key is executed (decryption, in 7 out of 16 types of cards) to verify key usability. 5) Operation with a public key is executed (encryption, 3 types of cards only).

Note that even when the key generation process is correctly guessed, it is not possible to simply implement it again and compare the resulting power traces – as only the card's main CPU is available for user-defined operations, instead of a coprocessor used by the original process. Additional side-channel and fault induction protection techniques may be also applied. Therefore, one cannot obtain an exactly matching power trace from a given card due to unavailability of low-level programming interfaces and additionally executed operations for verification of key generation hypothesis.

Whereas some steps of the key generation, such as the randomness generation, take an equal time across multiple runs of the process, the time required to generate a prime differs greatly as can be also seen from the example given in Figure 28, where timing is extracted from the power trace. The variability can be attributed to the randomized process of the prime generation. Incremental search will find the first prime greater than a random number selected as the base of the search. Since both primes p and q are distributed as distances from a random point to a prime number, the resulting time distribution will be affected by a mixture of these two distributions.

In samples collected from 12 out of 16 types of cards, the distribution of time is concentrated at evenly spaced points[13] as seen in Figure 28. The distance between a pair of points is interpreted as the duration of a single primality test, whereas their amount corresponds to the number of candidates that were ruled out by the test as a composite. Then it is possible to obtain a histogram of number of tested candidates, e.g., by binning the distribution with breaks placed in the midpoints of the empty intervals.

[13] Due to small differences in duration of key generation and rounding caused by precision of the measurement, the times belonging to the same group will not be identical to one millisecond. The peaks were highlighted by summing adjoining milliseconds, but only in the case when large (almost) empty spaces exist in the distribution.

8 Conclusions

This paper presents a thorough analysis of key pairs generated and extracted from 38 different sources encompassing open-source and proprietary software libraries and cryptographic cards. This broad analysis allowed us to assess current trends in RSA key pair generation even when the source codes for key generation were not available, as in the case of proprietary libraries and cards. The range of approaches identified indicates that the question of how to generate an optimal RSA key has not yet been settled.

The tested keys were generally found to contain a high level of entropy, sufficient to protect against known factorization attacks. However, the source-specific prime selection algorithms, postprocessing techniques and enforcement of specific properties (e.g., Blum primes) make the resulting primes slightly biased, and these biases serve as fingerprints of the sources. Our paper therefore shows that public moduli leak significantly more information than previously assumed. We identified seven properties of the generated primes that are propagated into the public moduli of the generated keys. As a result, accurate identification of originating library or smartcard is possible based only on knowledge of the public keys. Such an unexpected property can be used to decrease the anonymity set of RSA keys users, to search for keys generated by vulnerable libraries, to assess claims regarding the utilization of secure hardware by remote parties, and for other practical uses. We classified the probable origins of keys in two large datasets consisting of 10 and 15 million (mostly) TLS RSA keys and 1.4 million PGP RSA keys to obtain an estimate of the sources used in real-world applications.

The random number generator is a crucial component for the generation of strong keys. We identified a generic failure scenario that produces weak keys and occasionally detected such keys in our dataset obtained from the tested cards. Luckily, no such weak key was identified in the datasets of publicly used RSA keys.

Acknowledgements

We acknowledge the support of the Czech Science Foundation, project GA16-08565S. Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme “Projects of Large Research, Development, and Innovations Infrastructures” (CESNET LM2015042), is greatly appreciated.

We would like to thank all the anonymous reviewers of Usenix Security 2016 symposium for their helpful comments and fruitful discussions. Furthermore, we show our appreciation by thanking Martin Ukrop and Marek Sýs for reading the manuscript and providing insightful suggestions and L'ubomír Obrátil for running and interpreting statistical tests of randomness.

References

[1] Arma. Facebook, hidden services, and https certs, 2014. Tor blog, cit. [2016-06-26] <https://blog.torproject.org/blog/facebook-hidden-services-and-https-certs>.

[2] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. Efficient Padding Oracle Attacks on Cryptographic Hardware. In *Advances in Cryptology – CRYPTO 2012. Proceedings*, pages 608–625. Springer-Verlag, 2012.

[3] Luciano Bello. DSA-1571-1 openssl – predictable random number generator. Debian Security Advisory, 2008. Available from <https://www.debian.org/security/2008/dsa-1571>, cit. [2016-05-20].

[4] Daniel J. Bernstein, Yun-An Chang, Chen-Mou Cheng, Li-Ping Chou, Nadia Heninger, Tanja Lange, and Nicko Someren. Factoring RSA Keys from Certified Smart Cards: Coppersmith in the Wild. In *Advances in Cryptology – ASIACRYPT 2013. Proceedings*, pages 341–360. Springer-Verlag, 2013.

[5] Daniel J. Bernstein, Nadia Heninger, and Tanja Lange. Batch gcd, 2012. Available from <http://facthacks.cr.yp.to/batchgcd.html>, cit. [2016-05-20].

[6] Tom Berson, Drew Dean, Matt Franklin, Diana Smetters, and Michael Spreitzer. Cryptography as a network service. In *ISOC Network and Distributed System Security Symposium (NDSS). Proceedings*, 2001.

[7] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *Advances in Cryptology — CRYPTO '98. Proceedings*, pages 1–12. Springer-Verlag, 1998.

[8] Dan Boneh. Twenty Years of Attacks on the RSA Cryptosystem. *Notices of the AMS*, 46:203–213, 1999.

[9] Dan Boneh and Glenn Durfee. Cryptanalysis of RSA with Private Key d less than $N^{0.292}$. *IEEE Transactions on Information Theory*, 46(4):1339–1349, 2000.

[10] Ahmad Boorghany, Siavash Bayat Sarmadi, Pamian Yousefi, Pouneh Gorji, and Rasool Jalili. Random data and key generation evaluation of some commercial tokens and smart cards. In *Information Security and Cryptology (ISCISC), 11th International ISC Conference. Proceedings*, pages 49–54. IEEE, 2014.

[11] Robert G. Brown. Dieharder: A Random Number Test Suite, Version 3.31.1, 2004. Available from <http://www.phy.duke.edu/~rgb/General/dieharder.php>, cit. [2016-06-26].

[12] Billy Bob Brumley and Nicola Tuveri. Remote Timing Attacks Are Still Practical. In *Computer Security – ESORICS 2011. Proceedings*. Springer-Verlag, 2011.

[13] Don Coppersmith. Finding a Small Root of a Bivariate Integer Equation; Factoring with High Bits Known. In *15th Conference on Theory and Application of Cryptographic Techniques, EUROCRYPT'96. Proceedings*, pages 178–189. Springer-Verlag, 1996.

[14] Don Coppersmith. Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities. *Journal of Cryptology*, 10(4):233–260, 1997.

[15] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. A Search Engine Backed by Internet-Wide Scanning. In *22nd ACM Conference on Computer and Communications Security. Proceedings*, 2015. Dataset download: <https://scans.io/series/443-https-tls-full_ipv4>, [dataset from 2016-06-02].

[16] Google. Certificate Transparency dump from June 07, 2016 (log Pilot). Available from: <https://www.certificate-transparency.org/known-logs>, cit. [2016-06-07].

[17] John Gordon. Strong primes are easy to find. In *Advances in Cryptology: EUROCRYPT '84. Proceedings*, pages 216–223. Springer-Verlag, 1985.

[18] Internet Security Research Group. Let's Encrypt, 2016. Homepage: <https://letsencrypt.org/>, cit. [2016-06-26].

[19] Marc Gysin and Jennifer Seberry. Generalised Cycling Attacks on RSA and Strong RSA Primes. In *Information Security and Privacy: ACISP'99. Proceedings*, pages 149–163. Springer-Verlag, 1999.

[20] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *21st USENIX Security Symposium. Proceedings*, pages 205–220. USENIX, 2012.

[21] IEEE. Standard Specifications for Public-Key Cryptography. IEEE Std 1363, 2000.

[22] American National Standards Institute. ANSI X9.31-1998: Public Key Cryptography Using Reversible Algorithms for the Financial Services Industry (rDSA), 1998.

[23] Jakob Jonsson and Burton Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447, 2003.

[24] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-Bit RSA Modulus. In *Advances in Cryptology – CRYPTO 2010. Proceedings*, pages 333–350. Springer-Verlag, 2010.

[25] Donald E. Knuth and Luis Trabb Pardo. Analysis of a simple factorization algorithm. *Theoretical Computer Science*, 3(3):321 – 348, 1976.

[26] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Advances in Cryptology – CRYPTO'99. Proceedings*. Springer-Verlag, 1999.

[27] David Komárek. The RSA key generation process via power analysis. Master's thesis, 2016. Online: <https://is.muni.cz/th/395924/fi_m/>, cit. [2016-07-12].

[28] R. Sherman Lehman. Factoring large integers. In *Mathematics of Computation*, volume 28, pages 637–646. American Mathematical Society, 1974.

[29] Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung, and Christophe Wachter. Ron was wrong, Whit is right. *IACR Cryptology ePrint Archive*, 2012. Online: <http://eprint.iacr.org/2012/064>, cit. [2016-07-12].

[30] Hendrik W. Lenstra Jr. Factoring integers with Elliptic Curves. *The Annals of Mathematics*, 126:649–673, 1987.

[31] Daniel Loebenberger and Michael Nüsken. Notions for RSA Integers. In *International Journal of Applied Cryptology*, pages 116–138. Inderscience Publishers, 2014.

[32] Ueli M. Maurer. Fast generation of prime numbers and secure public-key cryptographic parameters. *Journal of Cryptology*, 8(3):123–155, 1995.

[33] Alfred J. Menezes, Paul C. Van Oorschot, Scott A. Vanstone, and Ronald L. Rivest. *Handbook of Applied Cryptography*. CRC Press, 1st edition, 1996.

[34] Ilya Mironov. Factoring RSA Moduli. Part II, 2012. Online, cit. [2016-06-26] <https://windowsontheory.org/2012/05/17/factoring-rsa-moduli-part-ii/>.

[35] Nano-Optics group and PicoQuant GmbH. High Bit Rate Quantum Random Number Generator Service. <http://qrng.physik.hu-berlin.de/>, cit. [2016-07-12].

[36] National Institute of Standards and Technology. Digital Signature Standard (DSS). FIPS 186-4, 2013.

[37] Matúš Nemec. The properties of RSA key generation process in software libraries. Master's thesis, 2016. Online: <https://is.muni.cz/th/396066/fi_m/>, cit. [2016-07-20].

[38] John M. Pollard. Theorems on factorization and primality testing. *Mathematical Proceedings of the Cambridge Philosophical Society*, 76:521–528, 1974.

[39] John M. Pollard. Factoring with cubic integers. In *The development of the number field sieve*, pages 4–10. Springer-Verlag, 1993.

[40] Carl Pomerance. The Quadratic Sieve Factoring Algorithm. In *Advances in Cryptology: EUROCRYPT '84. Proceedings*, pages 169–182. Springer-Verlag, 1985.

[41] Jurgen Pulkus. Efficient Prime-Number Check, 2014. US Patent App. 14/354,455.

[42] Jean-Jacques Quisquater and Chantal Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters*, 18(21):905–907, 1982.

[43] Tom Ritter. De-anonymizing alt.anonymous.messages. In *DEF CON 21 Archive*. DEF CON, 2013. Available from <https://ritter.vg/p/AAM-defcon13.pdf>, cit. [2016-06-26].

[44] Ronald L. Rivest and Robert D. Silverman. Are 'Strong' Primes Needed for RSA? In *The 1997 RSA Laboratories Seminar Series. Proceedings*, 1999.

[45] Phillip Rogaway and Stephen M. Matyas. RSAES-OAEP Encryption Scheme Algorithm specification and supporting documentation, 2000.

[46] Andrew Rukhin et al. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. In *NIST Special Publication 800-22rev1a*. NIST, 2010. Cit. [2016-02-18] <http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1a.pdf>.

[47] Robert D. Silverman. Fast Generation Of Random, Strong RSA Primes. In *CryptoBytes Technical Newsletter*. RSA Laboratories, 1997.

[48] Marek Sýs, Petr Švenda, Martin Ukrop, and Vashek Matyáš. Constructing empirical tests of randomness. In *SECRYPT 2014, SCITEPRESS*, pages 229–237, 2014.

[49] Petr Švenda, Matúš Nemec, Peter Sekan, Rudolf Kvašňovský, David Formánek, David Komárek, and Vashek Matyáš. The Million-Key Question – Investigating the Origins of RSA Public Keys. In *25th USENIX Security Symposium. Proceedings*, 2016.

[50] David Wagner. Cryptanalysis of a Provably Secure CRT-RSA Algorithm. In *11th ACM Conference on Computer and Communications Security. Proceedings*, pages 92–97. ACM, 2004.

[51] Michael J. Wiener. Cryptanalysis of short RSA secret exponents. In *IEEE Transactions on Information Theory*, volume 36, pages 553–558. IEEE, 1990.

[52] Hugh C. Williams. A $p + 1$ Method of Factoring. In *Mathematics of Computation*, volume 39, pages 225–234. American Mathematical Society, 1982.

[53] YAFU: Yet Another Factorization Utility, 2013. Available from <http://sourceforge.net/projects/yafu/>, cit. [2016-06-26].

[54] PGP keydump from June 02, 2016. Available from <http://pgp.key-server.io/dump/current/>, cit. [2016-06-02].

[55] Keys collected in the 1MRSA project, 2016. Available from <http://crcs.cz/papers/usenix2016/1mrsaset>.

[56] W3Techs Web Technology Surveys: Usage of web servers for websites, 2016. <http://w3techs.com/technologies/overview/web_server/all>, cit. [2016-06-26].

A Tested corrupted keys

The public keys from TLS IPv4 [15] and PGP RSA keys dataset [54] were divided by the following primes, obtained by the faulty algorithm as described in Section 7.1, however none of the keys had such factor.

```
0x80000000000000000000000000000000000000000000000000000000000005f
0x9000000000000000000000000000000000000000000000000000000000000145
0x9000000000000000000000000000000000000000000000000000000000000253
0xa00000000000000000000000000000000000000000000000000000000000009d
0xa0000000000000000000000000000000000000000000000000000000000000f3
0xb0000000000000000000000000000000000000000000000000000000000000c9
0xb0000000000000000000000000000000000000000000000000000000000000fb
0xc000000000000000000000000000000000000000000000000000000000000031
0xc000000000000000000000000000000000000000000000000000000000000077
0xd00000000000000000000000000000000000000000000000000000000000004f
0xe0000000000000000000000000000000000000000000000000000000000001ad
0xe000000000000000000000000000000000000000000000000000000000000233
0xf00000000000000000000000000000000000000000000000000000000000013d
0xf0000000000000000000000000000000000000000000000000000000000001cf
0x8000000000000000000000000000000000000000000000000000000000000000
  000000000000000000000000000000000000000000000000000000000000006f
0x9000000000000000000000000000000000000000000000000000000000000000
  000000000000000000000000000000000000000000000000000000000000001d
0x9000000000000000000000000000000000000000000000000000000000000000
  00000000000000000000000000000000000000000000000000000000000000cb
0xa000000000000000000000000000000000000000000000000000000000000000
  00000000000000000000000000000000000000000000000000000000000000b5
0xa000000000000000000000000000000000000000000000000000000000000000
  000000000000000000000000000000000000000000000000000000000000061f
0xb000000000000000000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000000000000000000000000000000015
0xb000000000000000000000000000000000000000000000000000000000000000
  000000000000000000000000000000000000000000000000000000000000006b
0xc000000000000000000000000000000000000000000000000000000000000000
```

```
00000000000000000000000000000000000000000000000000000000000002f9
0xc000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000e9b
0xd000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000135
0xd000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000032b
0xe000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000f
0xf000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000007f
0x8000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000483
0x9000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000002ab
0xa000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000ab
0xb000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000003f
0xc000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000040d
0xc000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
```

```
  0000000000000000000000000000000000000000000000000000000000000827
0xd000000000000000000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000000000000000000000000000000465
0xd000000000000000000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000000000000000000000000000000000
  00000000000000000000000000000000000000000000000000000000000004e3
0xe000000000000000000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000000000000000000000000000000135
0xe000000000000000000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000000000000000000000000000000000
  00000000000000000000000000000000000000000000000000000000000005cf
0xf000000000000000000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000000000000000000000000000000000
  000000000000000000000000000000000000000000000000000000000000065b
```

B Classification matrix for 13 groups of sources

We present the classification matrix built according to the algorithm described in Section 5.1 for 13 distinct groups formed from 38 source libraries and smartcards.

All modulus bits identified through previous analysis as non-uniform for at least one source are included in a *mask*. The mask values are arranged in the following order: the values of the $2^{\text{nd}} - 7^{\text{th}}$ most significant bits influenced by the prime manipulations described in Section 4.1, the second least significant bit (which is always zero for sources that use Blum integers), the result of the modulus modulo 3 (which is influenced by the avoidance of factor 3 in $\mathtt{p}-1$ and $\mathtt{q}-1$) and the overall modulus length modulo 2 (which indicates whether an exact length is enforced).

Mask value	Group of sources												
	I	II	III	IV	V	VI	VII	VIII	IX	X	XI	XII	XIII
000000 ∣ 0 ∣ 1 ∣ 0	-	52.49%	33.41%	2.49%	-	-	-	11.43%	-	0.18%	-	-	-
000000 ∣ 0 ∣ 1 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000000 ∣ 0 ∣ 2 ∣ 0	-	79.05%	-	3.67%	-	-	-	17.01%	-	0.26%	-	-	-
000000 ∣ 0 ∣ 2 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000000 ∣ 1 ∣ 1 ∣ 0	-	-	-	16.92%	-	-	-	81.78%	-	1.29%	-	-	-
000000 ∣ 1 ∣ 1 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000000 ∣ 1 ∣ 2 ∣ 0	-	0.01%	-	17.55%	-	-	-	81.16%	-	1.27%	-	-	-
000000 ∣ 1 ∣ 2 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000001 ∣ 0 ∣ 1 ∣ 0	-	36.73%	49.37%	2.10%	-	-	0.60%	10.62%	-	0.59%	-	-	-
000001 ∣ 0 ∣ 1 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000001 ∣ 0 ∣ 2 ∣ 0	-	72.30%	-	4.16%	-	-	1.18%	21.21%	-	1.15%	-	-	-
000001 ∣ 0 ∣ 2 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000001 ∣ 1 ∣ 1 ∣ 0	-	-	-	15.91%	-	-	-	79.73%	-	4.36%	-	-	-
000001 ∣ 1 ∣ 1 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000001 ∣ 1 ∣ 2 ∣ 0	-	-	-	15.34%	-	-	-	80.22%	-	4.44%	-	-	-
000001 ∣ 1 ∣ 2 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000010 ∣ 0 ∣ 1 ∣ 0	-	33.29%	50.31%	1.98%	-	-	2.47%	10.90%	0.01%	1.05%	-	-	-
000010 ∣ 0 ∣ 1 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000010 ∣ 0 ∣ 2 ∣ 0	-	66.68%	-	3.92%	-	-	4.98%	22.25%	-	2.17%	-	-	-
000010 ∣ 0 ∣ 2 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000010 ∣ 1 ∣ 1 ∣ 0	-	-	-	14.39%	-	-	-	77.76%	0.07%	7.77%	-	-	-
000010 ∣ 1 ∣ 1 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000010 ∣ 1 ∣ 2 ∣ 0	-	-	-	13.74%	-	-	-	78.61%	-	7.66%	-	-	-
000010 ∣ 1 ∣ 2 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000011 ∣ 0 ∣ 1 ∣ 0	4.55%	28.29%	46.83%	1.67%	-	-	4.80%	10.19%	-	1.42%	-	-	2.27%
000011 ∣ 0 ∣ 1 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000011 ∣ 0 ∣ 2 ∣ 0	-	57.88%	-	3.39%	-	-	9.95%	21.13%	0.01%	2.96%	-	-	4.68%
000011 ∣ 0 ∣ 2 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000011 ∣ 1 ∣ 1 ∣ 0	-	-	-	12.92%	-	-	-	76.25%	0.04%	10.79%	-	-	-
000011 ∣ 1 ∣ 1 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000011 ∣ 1 ∣ 2 ∣ 0	-	-	-	12.75%	-	-	-	76.55%	-	10.70%	-	-	-
000011 ∣ 1 ∣ 2 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000100 ∣ 0 ∣ 1 ∣ 0	24.65%	17.61%	30.81%	1.04%	-	-	5.49%	6.79%	0.00%	1.26%	-	-	12.37%
000100 ∣ 0 ∣ 1 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000100 ∣ 0 ∣ 2 ∣ 0	-	39.31%	-	2.15%	-	-	12.37%	15.38%	-	2.86%	-	-	27.93%
000100 ∣ 0 ∣ 2 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000100 ∣ 1 ∣ 1 ∣ 0	-	-	-	10.64%	-	-	-	75.09%	0.19%	14.09%	-	-	-
000100 ∣ 1 ∣ 1 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000100 ∣ 1 ∣ 2 ∣ 0	-	-	-	10.97%	-	-	-	74.98%	0.26%	13.79%	-	-	-
000100 ∣ 1 ∣ 2 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000101 ∣ 0 ∣ 1 ∣ 0	35.89%	11.82%	22.06%	0.67%	-	-	5.70%	4.86%	0.01%	1.11%	-	-	17.88%
000101 ∣ 0 ∣ 1 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000101 ∣ 0 ∣ 2 ∣ 0	-	28.18%	-	1.47%	-	-	13.43%	11.51%	0.05%	2.70%	-	-	42.66%
000101 ∣ 0 ∣ 2 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000101 ∣ 1 ∣ 1 ∣ 0	-	-	-	9.90%	-	-	-	72.86%	0.26%	16.99%	-	-	-
000101 ∣ 1 ∣ 1 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000101 ∣ 1 ∣ 2 ∣ 0	-	-	-	10.05%	-	-	-	72.67%	0.48%	16.80%	-	-	-
000101 ∣ 1 ∣ 2 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000110 ∣ 0 ∣ 1 ∣ 0	42.24%	8.68%	16.95%	0.45%	-	-	5.78%	3.76%	0.04%	1.05%	-	-	21.06%
000110 ∣ 0 ∣ 1 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000110 ∣ 0 ∣ 2 ∣ 0	-	21.31%	-	1.13%	-	-	14.09%	9.17%	0.09%	2.58%	-	-	51.63%
000110 ∣ 0 ∣ 2 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000110 ∣ 1 ∣ 1 ∣ 0	-	-	-	8.39%	-	-	-	71.07%	0.55%	19.98%	-	-	-
000110 ∣ 1 ∣ 1 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000110 ∣ 1 ∣ 2 ∣ 0	-	-	-	8.29%	-	-	-	71.18%	0.55%	19.98%	-	-	-
000110 ∣ 1 ∣ 2 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000111 ∣ 0 ∣ 1 ∣ 0	46.37%	6.74%	13.65%	0.33%	-	-	5.78%	3.03%	0.05%	1.00%	-	-	23.05%
000111 ∣ 0 ∣ 1 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000111 ∣ 0 ∣ 2 ∣ 0	-	16.67%	-	0.79%	-	-	14.41%	7.54%	0.13%	2.46%	-	-	57.99%
000111 ∣ 0 ∣ 2 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000111 ∣ 1 ∣ 1 ∣ 0	-	0.01%	-	7.82%	-	-	-	68.17%	1.76%	22.25%	-	-	-
000111 ∣ 1 ∣ 1 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
000111 ∣ 1 ∣ 2 ∣ 0	-	-	-	7.42%	-	-	-	68.37%	1.87%	22.33%	-	-	-
000111 ∣ 1 ∣ 2 ∣ 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-

Table 12: *The classification table, part 1 of 8. The mask is arranged as:* $2^{\text{nd}}-7^{\text{th}}$ *most significant bit of modulus | 2^{nd} least significant bit of modulus | modulus mod 3 | modulus length mod 2.*

Mask value	Group of sources												
	I	II	III	IV	V	VI	VII	VIII	IX	X	XI	XII	XIII
001000 | 0 | 1 | 0	46.57%	5.76%	11.88%	0.26%	0.16%	0.17%	5.59%	2.67%	0.11%	1.01%	0.07%	2.61%	23.15%
001000 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001000 | 0 | 2 | 0	-	13.84%	-	0.66%	-	0.39%	13.46%	6.43%	0.27%	2.45%	0.16%	6.36%	55.97%
001000 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001000 | 1 | 1 | 0	-	-	-	3.85%	2.36%	-	-	38.88%	1.48%	14.57%	0.96%	37.89%	-
001000 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001000 | 1 | 2 | 0	-	-	-	3.85%	-	-	-	39.81%	1.68%	14.95%	1.01%	38.71%	-
001000 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001001 | 0 | 1 | 0	44.02%	5.11%	10.99%	0.21%	0.45%	0.45%	5.27%	2.46%	0.15%	1.07%	0.19%	7.44%	22.20%
001001 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001001 | 0 | 2 | 0	-	11.45%	-	0.49%	-	0.98%	11.84%	5.56%	0.40%	2.40%	0.42%	16.62%	49.84%
001001 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001001 | 1 | 1 | 0	-	-	-	1.73%	3.74%	-	-	20.84%	1.33%	8.95%	1.57%	61.83%	-
001001 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001001 | 1 | 2 | 0	-	-	-	1.86%	-	-	-	21.51%	1.55%	9.33%	1.62%	64.13%	-
001001 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001010 | 0 | 1 | 0	42.09%	4.60%	10.13%	0.19%	0.68%	0.68%	5.04%	2.30%	0.23%	1.12%	0.32%	11.62%	21.01%
001010 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001010 | 0 | 2 | 0	-	9.74%	-	0.42%	-	1.50%	10.63%	4.89%	0.54%	2.35%	0.66%	24.63%	44.64%
001010 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001010 | 1 | 1 | 0	-	-	-	1.14%	4.26%	-	-	14.03%	1.45%	6.73%	1.93%	70.46%	-
001010 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001010 | 1 | 2 | 0	-	-	-	1.15%	-	-	-	14.61%	1.61%	7.03%	2.01%	73.58%	-
001010 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001011 | 0 | 1 | 0	38.83%	4.50%	10.11%	0.17%	1.00%	1.01%	5.02%	2.31%	0.38%	1.24%	0.47%	15.62%	19.34%
001011 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001011 | 0 | 2 | 0	-	8.99%	-	0.33%	-	2.00%	10.04%	4.60%	0.77%	2.50%	0.94%	30.94%	38.89%
001011 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001011 | 1 | 1 | 0	-	-	-	0.77%	4.68%	-	-	10.90%	1.90%	5.85%	2.22%	73.68%	-
001011 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001011 | 1 | 2 | 0	-	-	-	0.78%	-	-	-	11.51%	2.02%	6.19%	2.29%	77.21%	-
001011 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001100 | 0 | 1 | 0	34.60%	4.93%	11.35%	0.15%	1.47%	1.42%	5.46%	2.60%	0.61%	1.56%	0.71%	17.96%	17.19%
001100 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001100 | 0 | 2 | 0	-	9.28%	-	0.29%	-	2.86%	10.37%	4.93%	1.32%	2.95%	1.32%	33.98%	32.69%
001100 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001100 | 1 | 1 | 0	-	0.00%	-	0.65%	5.79%	-	-	10.43%	2.77%	6.25%	2.77%	71.34%	-
001100 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001100 | 1 | 2 | 0	-	-	-	0.75%	-	-	-	11.10%	2.74%	6.59%	2.94%	75.88%	-
001100 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001101 | 0 | 1 | 0	29.72%	5.72%	13.43%	0.19%	2.14%	2.13%	6.27%	3.09%	1.10%	2.04%	1.05%	18.37%	14.75%
001101 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001101 | 0 | 2 | 0	-	10.39%	-	0.33%	-	3.87%	11.44%	5.61%	1.90%	3.68%	1.91%	33.36%	27.50%
001101 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001101 | 1 | 1 | 0	-	-	-	0.66%	7.76%	-	-	11.07%	3.91%	7.25%	3.75%	65.60%	-
001101 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001101 | 1 | 2 | 0	-	-	-	0.70%	-	-	-	11.98%	4.27%	7.87%	4.05%	71.12%	-
001101 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001110 | 0 | 1 | 0	23.21%	7.05%	16.81%	0.20%	3.27%	3.33%	7.95%	3.90%	1.99%	2.83%	1.61%	16.38%	11.46%
001110 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001110 | 0 | 2 | 0	-	12.41%	-	0.36%	-	5.91%	14.08%	6.95%	3.13%	5.00%	2.87%	29.07%	20.22%
001110 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001110 | 1 | 1 | 0	-	-	-	0.64%	11.04%	-	-	13.13%	5.74%	9.42%	5.38%	54.64%	-
001110 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001110 | 1 | 2 | 0	-	-	-	0.77%	-	-	-	14.54%	7.30%	10.58%	5.99%	60.83%	-
001110 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001111 | 0 | 1 | 0	10.71%	9.47%	22.90%	0.24%	5.30%	5.36%	11.19%	5.36%	3.54%	4.21%	2.63%	13.49%	5.60%
001111 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001111 | 0 | 2 | 0	-	15.64%	-	0.43%	-	8.90%	18.35%	8.85%	5.86%	6.94%	4.33%	21.99%	8.71%
001111 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001111 | 1 | 1 | 0	-	-	-	0.70%	15.54%	-	-	15.74%	9.73%	12.29%	7.58%	38.42%	-
001111 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
001111 | 1 | 2 | 0	-	-	-	0.92%	-	-	-	18.12%	11.78%	14.26%	8.92%	46.00%	-
001111 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-

Table 13: *The classification table, part 2 of 8. The mask is arranged as:* $2^{\text{nd}}-7^{\text{th}}$ *most significant bit of modulus* | 2^{nd} *least significant bit of modulus* | *modulus mod 3* | *modulus length mod 2.*

Mask value	Group of sources												
	I	II	III	IV	V	VI	VII	VIII	IX	X	XI	XII	XIII
010000 | 0 | 1 | 0	-	11.70%	28.44%	0.30%	7.62%	7.61%	14.26%	6.66%	5.75%	5.74%	3.77%	8.16%	-
010000 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010000 | 0 | 2 | 0	-	18.25%	-	0.42%	-	12.01%	22.43%	10.48%	8.66%	9.02%	5.98%	12.76%	-
010000 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010000 | 1 | 1 | 0	-	-	-	0.71%	20.38%	-	-	17.66%	14.28%	15.13%	10.12%	21.72%	-
010000 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010000 | 1 | 2 | 0	-	-	-	0.96%	-	-	-	22.03%	18.43%	18.86%	12.55%	27.18%	-
010000 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010001 | 0 | 1 | 0	-	11.43%	28.60%	0.23%	8.75%	8.80%	14.70%	6.68%	7.17%	6.23%	4.36%	3.04%	-
010001 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010001 | 0 | 2 | 0	-	18.36%	-	0.43%	-	13.85%	23.59%	10.80%	10.97%	9.97%	7.02%	5.00%	-
010001 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010001 | 1 | 1 | 0	-	-	-	0.67%	24.09%	-	-	18.50%	19.57%	16.83%	11.98%	8.37%	-
010001 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010001 | 1 | 2 | 0	-	-	-	0.90%	-	-	-	24.23%	25.42%	22.54%	15.83%	11.08%	-
010001 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010010 | 0 | 1 | 0	-	10.85%	27.66%	0.23%	9.63%	9.80%	14.76%	6.59%	8.30%	6.52%	4.82%	0.85%	-
010010 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010010 | 0 | 2 | 0	-	17.79%	-	0.37%	-	15.04%	23.62%	10.50%	13.10%	10.51%	7.70%	1.37%	-
010010 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010010 | 1 | 1 | 0	-	-	-	0.67%	26.67%	-	-	17.90%	21.39%	17.90%	13.15%	2.31%	-
010010 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010010 | 1 | 2 | 0	-	-	-	0.85%	-	-	-	24.08%	30.34%	24.05%	17.52%	3.16%	-
010010 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010011 | 0 | 1 | 0	-	10.23%	26.24%	0.20%	10.42%	10.68%	14.42%	6.23%	9.53%	6.78%	5.16%	0.11%	-
010011 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010011 | 0 | 2 | 0	-	16.26%	-	0.33%	-	16.75%	22.93%	9.87%	14.89%	10.67%	8.15%	0.16%	-
010011 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010011 | 1 | 1 | 0	-	-	-	0.53%	27.16%	-	-	16.43%	24.34%	17.73%	13.53%	0.29%	-
010011 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010011 | 1 | 2 | 0	-	-	-	0.71%	-	-	-	22.67%	33.25%	24.38%	18.62%	0.38%	-
010011 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010100 | 0 | 1 | 0	-	9.73%	24.97%	0.18%	11.01%	11.17%	14.15%	5.93%	10.42%	6.92%	5.51%	0.00%	-
010100 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010100 | 0 | 2 | 0	-	15.32%	-	0.26%	-	17.13%	22.19%	9.33%	16.25%	10.87%	8.65%	0.00%	-
010100 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010100 | 1 | 1 | 0	-	0.00%	-	0.40%	27.19%	-	-	14.94%	26.58%	17.23%	13.66%	0.00%	-
010100 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010100 | 1 | 2 | 0	-	-	-	0.61%	-	-	-	20.27%	37.01%	23.48%	18.63%	0.00%	-
010100 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010101 | 0 | 1 | 0	-	9.23%	23.89%	0.16%	11.48%	11.88%	13.25%	5.70%	11.45%	7.10%	5.86%	-	-
010101 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010101 | 0 | 2 | 0	-	14.49%	-	0.23%	-	18.40%	20.61%	8.88%	17.34%	10.97%	9.08%	-	-
010101 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010101 | 1 | 1 | 0	-	-	-	0.36%	27.62%	-	-	13.48%	28.08%	16.75%	13.70%	-	-
010101 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010101 | 1 | 2 | 0	-	-	-	0.49%	-	-	-	18.98%	37.39%	23.63%	19.51%	-	-
010101 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010110 | 0 | 1 | 0	-	8.63%	22.68%	0.15%	12.22%	12.70%	12.38%	5.49%	12.22%	7.34%	6.19%	-	-
010110 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010110 | 0 | 2 | 0	-	13.55%	-	0.24%	-	19.11%	19.13%	8.48%	18.64%	11.34%	9.51%	-	-
010110 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010110 | 1 | 1 | 0	-	-	-	0.32%	28.41%	-	-	12.60%	27.46%	16.95%	14.26%	-	-
010110 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010110 | 1 | 2 | 0	-	-	-	0.42%	-	-	-	17.38%	39.20%	23.42%	19.59%	-	-
010110 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010111 | 0 | 1 | 0	-	8.37%	21.93%	0.13%	12.89%	12.85%	11.45%	5.21%	13.13%	7.53%	6.51%	-	-
010111 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010111 | 0 | 2 | 0	-	12.95%	-	0.20%	-	20.14%	17.46%	8.00%	19.79%	11.56%	9.90%	-	-
010111 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010111 | 1 | 1 | 0	-	-	-	0.30%	28.33%	-	-	11.49%	29.10%	16.52%	14.26%	-	-
010111 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
010111 | 1 | 2 | 0	-	-	-	0.40%	-	-	-	16.09%	40.39%	23.16%	19.96%	-	-
010111 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-

Table 14: *The classification table, part 3 of 8. The mask is arranged as: $2^{\text{nd}} - 7^{\text{th}}$ most significant bit of modulus | 2^{nd} least significant bit of modulus | modulus mod 3 | modulus length mod 2.*

Mask value	Group of sources												
	I	II	III	IV	V	VI	VII	VIII	IX	X	XI	XII	XIII
011000∣0∣1∣0	-	8.07%	20.86%	0.11%	13.64%	13.49%	10.65%	5.03%	13.42%	7.84%	6.90%	-	-
011000∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011000∣0∣2∣0	-	12.12%	-	0.17%	-	21.21%	16.31%	7.70%	20.11%	11.95%	10.44%	-	-
011000∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011000∣1∣1∣0	-	-	-	0.26%	29.12%	-	-	10.84%	28.32%	16.79%	14.67%	-	-
011000∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011000∣1∣2∣0	-	-	-	0.34%	-	-	-	15.26%	40.21%	23.42%	20.77%	-	-
011000∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011001∣0∣1∣0	-	7.53%	19.80%	0.13%	14.45%	14.46%	9.91%	4.88%	13.52%	8.09%	7.24%	-	-
011001∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011001∣0∣2∣0	-	11.62%	-	0.18%	-	22.03%	15.07%	7.31%	20.67%	12.16%	10.96%	-	-
011001∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011001∣1∣1∣0	-	-	-	0.23%	29.73%	-	-	9.97%	28.36%	16.70%	15.00%	-	-
011001∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011001∣1∣2∣0	-	-	-	0.34%	-	-	-	14.30%	40.37%	23.78%	21.21%	-	-
011001∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011010∣0∣1∣0	-	7.18%	18.98%	0.12%	15.09%	15.41%	9.17%	4.69%	13.51%	8.24%	7.60%	-	-
011010∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011010∣0∣2∣0	-	10.95%	-	0.15%	-	22.60%	14.03%	7.11%	20.93%	12.67%	11.56%	-	-
011010∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011010∣1∣1∣0	-	-	-	0.23%	30.33%	-	-	9.52%	27.75%	16.78%	15.40%	-	-
011010∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011010∣1∣2∣0	-	-	-	0.34%	-	-	-	13.79%	39.16%	24.19%	22.52%	-	-
011010∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011011∣0∣1∣0	-	6.98%	18.28%	0.10%	15.76%	15.66%	8.78%	4.49%	13.97%	8.06%	7.91%	-	-
011011∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011011∣0∣2∣0	-	10.30%	-	0.15%	-	24.02%	13.29%	6.80%	21.23%	12.19%	12.03%	-	-
011011∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011011∣1∣1∣0	-	-	-	0.23%	31.19%	-	-	8.85%	28.04%	15.97%	15.73%	-	-
011011∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011011∣1∣2∣0	-	-	-	0.32%	-	-	-	13.02%	40.21%	23.50%	22.95%	-	-
011011∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011100∣0∣1∣0	-	6.64%	17.60%	0.10%	16.34%	16.68%	8.76%	4.28%	13.44%	7.83%	8.33%	-	-
011100∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011100∣0∣2∣0	-	10.21%	-	0.15%	-	24.96%	13.28%	6.53%	20.51%	11.83%	12.54%	-	-
011100∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011100∣1∣1∣0	-	-	-	0.22%	32.69%	-	-	8.47%	26.67%	15.41%	16.55%	-	-
011100∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011100∣1∣2∣0	-	-	-	0.31%	-	-	-	12.68%	39.68%	22.98%	24.34%	-	-
011100∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011101∣0∣1∣0	-	6.24%	16.82%	0.12%	17.16%	17.51%	8.78%	4.12%	13.08%	7.56%	8.62%	-	-
011101∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011101∣0∣2∣0	-	9.76%	-	0.21%	-	26.30%	13.40%	6.31%	19.30%	11.53%	13.20%	-	-
011101∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011101∣1∣1∣0	-	-	-	0.21%	34.20%	-	-	8.22%	25.41%	14.93%	17.02%	-	-
011101∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011101∣1∣2∣0	-	-	-	0.36%	-	-	-	12.52%	37.91%	22.99%	26.21%	-	-
011101∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011110∣0∣1∣0	-	6.11%	16.23%	0.12%	18.02%	18.00%	8.89%	4.04%	12.19%	7.32%	9.09%	-	-
011110∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011110∣0∣2∣0	-	9.28%	-	0.18%	-	27.12%	13.53%	6.08%	18.79%	11.21%	13.80%	-	-
011110∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011110∣1∣1∣0	-	-	-	0.25%	35.25%	-	-	7.77%	24.50%	14.40%	17.84%	-	-
011110∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011110∣1∣2∣0	-	-	-	0.43%	-	-	-	12.14%	37.59%	22.27%	27.56%	-	-
011110∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011111∣0∣1∣0	-	5.89%	15.39%	0.14%	18.77%	18.83%	8.86%	3.85%	11.62%	7.14%	9.51%	-	-
011111∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011111∣0∣2∣0	-	9.18%	-	0.18%	-	28.65%	13.53%	5.90%	17.24%	10.81%	14.53%	-	-
011111∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011111∣1∣1∣0	-	-	-	0.27%	37.08%	-	-	7.64%	22.35%	14.00%	18.67%	-	-
011111∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
011111∣1∣2∣0	-	-	-	0.43%	-	-	-	12.12%	35.71%	22.22%	29.52%	-	-
011111∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-

Table 15: *The classification table, part 4 of 8. The mask is arranged as:* $2^{\mathrm{nd}} - 7^{\mathrm{th}}$ *most significant bit of modulus* ∣ 2^{nd} *least significant bit of modulus* ∣ *modulus mod 3* ∣ *modulus length mod 2.*

Mask value	Group of sources												
	I	II	III	IV	V	VI	VII	VIII	IX	X	XI	XII	XIII
100000 | 0 | 1 | 0	-	5.92%	15.30%	0.16%	19.28%	19.20%	9.17%	3.83%	10.33%	7.07%	9.74%	-	-
100000 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100000 | 0 | 2 | 0	-	8.94%	-	0.22%	-	29.78%	13.78%	5.80%	15.95%	10.75%	14.78%	-	-
100000 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100000 | 1 | 1 | 0	-	-	-	0.29%	37.83%	-	-	7.51%	21.38%	13.83%	19.16%	-	-
100000 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100000 | 1 | 2 | 0	-	-	-	0.42%	-	-	-	12.20%	33.94%	22.40%	31.04%	-	-
100000 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100001 | 0 | 1 | 0	-	5.74%	15.11%	0.17%	19.43%	19.20%	9.58%	3.78%	10.07%	7.13%	9.78%	-	-
100001 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100001 | 0 | 2 | 0	-	8.77%	-	0.25%	-	29.43%	14.53%	5.79%	15.45%	10.84%	14.93%	-	-
100001 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100001 | 1 | 1 | 0	-	-	-	0.36%	38.45%	-	-	7.63%	20.13%	14.06%	19.36%	-	-
100001 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100001 | 1 | 2 | 0	-	-	-	0.55%	-	-	-	12.43%	32.41%	22.94%	31.67%	-	-
100001 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100010 | 0 | 1 | 0	-	5.67%	14.93%	0.19%	19.45%	19.09%	10.30%	3.79%	9.79%	7.05%	9.75%	-	-
100010 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100010 | 0 | 2 | 0	-	8.76%	-	0.34%	-	28.99%	15.68%	5.79%	14.60%	10.88%	14.96%	-	-
100010 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100010 | 1 | 1 | 0	-	-	-	0.43%	39.25%	-	-	7.57%	18.80%	14.30%	19.65%	-	-
100010 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100010 | 1 | 2 | 0	-	-	-	0.73%	-	-	-	12.57%	31.48%	23.33%	31.88%	-	-
100010 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100011 | 0 | 1 | 0	-	5.60%	14.78%	0.22%	19.46%	19.62%	10.60%	3.76%	9.13%	7.10%	9.73%	-	-
100011 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100011 | 0 | 2 | 0	-	8.56%	-	0.36%	-	29.62%	16.34%	5.85%	13.47%	10.82%	14.98%	-	-
100011 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100011 | 1 | 1 | 0	-	-	-	0.47%	39.57%	-	-	7.72%	17.83%	14.55%	19.86%	-	-
100011 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100011 | 1 | 2 | 0	-	-	-	0.82%	-	-	-	12.75%	29.06%	24.22%	33.15%	-	-
100011 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100100 | 0 | 1 | 0	-	5.81%	14.45%	0.28%	19.63%	19.46%	11.36%	3.82%	8.00%	7.23%	9.96%	-	-
100100 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100100 | 0 | 2 | 0	-	8.97%	-	0.42%	-	29.80%	17.13%	5.88%	11.69%	10.93%	15.18%	-	-
100100 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100100 | 1 | 1 | 0	-	-	-	0.59%	40.46%	-	-	7.88%	15.88%	14.79%	20.40%	-	-
100100 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100100 | 1 | 2 | 0	-	-	-	0.90%	-	-	-	13.35%	26.60%	24.89%	34.26%	-	-
100100 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100101 | 0 | 1 | 0	-	5.70%	14.23%	0.30%	19.95%	19.92%	11.87%	3.88%	6.95%	7.21%	10.00%	-	-
100101 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100101 | 0 | 2 | 0	-	8.48%	-	0.45%	-	29.96%	18.05%	5.96%	10.71%	11.16%	15.23%	-	-
100101 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100101 | 1 | 1 | 0	-	-	-	0.64%	41.04%	-	-	7.96%	14.48%	15.04%	20.84%	-	-
100101 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100101 | 1 | 2 | 0	-	-	-	1.05%	-	-	-	13.45%	25.19%	25.43%	34.87%	-	-
100101 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100110 | 0 | 1 | 0	-	5.76%	13.94%	0.37%	19.94%	20.05%	12.48%	3.87%	6.18%	7.30%	10.10%	-	-
100110 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100110 | 0 | 2 | 0	-	8.64%	-	0.57%	-	30.30%	18.82%	5.79%	9.70%	11.04%	15.14%	-	-
100110 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100110 | 1 | 1 | 0	-	-	-	0.75%	41.20%	-	-	8.03%	13.65%	15.33%	21.03%	-	-
100110 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100110 | 1 | 2 | 0	-	-	-	1.33%	-	-	-	13.74%	23.02%	26.15%	35.75%	-	-
100110 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100111 | 0 | 1 | 0	-	5.66%	13.86%	0.41%	20.07%	20.01%	13.06%	3.83%	5.67%	7.39%	10.05%	-	-
100111 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100111 | 0 | 2 | 0	-	8.49%	-	0.65%	-	30.11%	19.64%	5.85%	8.97%	11.04%	15.25%	-	-
100111 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100111 | 1 | 1 | 0	-	-	-	0.86%	42.38%	-	-	8.14%	11.52%	15.68%	21.42%	-	-
100111 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
100111 | 1 | 2 | 0	-	-	-	1.60%	-	-	-	14.07%	20.96%	26.58%	36.78%	-	-
100111 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-

Table 16: *The classification table, part 5 of 8. The mask is arranged as:* $2^{\mathrm{nd}} - 7^{\mathrm{th}}$ *most significant bit of modulus* | 2^{nd} *least significant bit of modulus* | *modulus mod 3* | *modulus length mod 2.*

Mask value	Group of sources												
	I	II	III	IV	V	VI	VII	VIII	IX	X	XI	XII	XIII
101000 | 0 | 1 | 0	-	5.60%	13.61%	0.52%	20.37%	20.05%	13.37%	3.92%	4.83%	7.45%	10.27%	-	-
101000 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101000 | 0 | 2 | 0	-	8.77%	-	0.76%	-	29.73%	20.47%	5.89%	7.50%	11.27%	15.62%	-	-
101000 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101000 | 1 | 1 | 0	-	-	-	1.07%	43.32%	-	-	8.26%	10.08%	15.64%	21.63%	-	-
101000 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101000 | 1 | 2 | 0	-	-	-	1.76%	-	-	-	14.57%	17.29%	27.93%	38.43%	-	-
101000 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101001 | 0 | 1 | 0	-	5.73%	13.53%	0.58%	20.41%	19.96%	13.83%	3.95%	4.14%	7.52%	10.34%	-	-
101001 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101001 | 0 | 2 | 0	-	8.57%	-	0.79%	-	30.58%	20.76%	5.98%	6.23%	11.43%	15.66%	-	-
101001 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101001 | 1 | 1 | 0	-	-	-	1.22%	43.89%	-	-	8.26%	8.74%	15.99%	21.90%	-	-
101001 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101001 | 1 | 2 | 0	-	-	-	2.11%	-	-	-	14.94%	15.17%	28.53%	39.24%	-	-
101001 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101010 | 0 | 1 | 0	-	5.83%	13.09%	0.69%	20.85%	19.71%	14.51%	3.95%	3.37%	7.62%	10.38%	-	-
101010 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101010 | 0 | 2 | 0	-	8.63%	-	1.01%	-	30.80%	21.85%	5.88%	4.92%	11.33%	15.58%	-	-
101010 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101010 | 1 | 1 | 0	-	-	-	1.49%	44.27%	-	-	8.47%	7.09%	16.32%	22.36%	-	-
101010 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101010 | 1 | 2 | 0	-	-	-	2.60%	-	-	-	15.07%	13.91%	28.73%	39.69%	-	-
101010 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101011 | 0 | 1 | 0	-	5.77%	12.58%	0.76%	20.76%	20.10%	15.26%	3.94%	2.87%	7.58%	10.37%	-	-
101011 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101011 | 0 | 2 | 0	-	8.66%	-	1.21%	-	30.61%	22.81%	5.82%	4.09%	11.35%	15.46%	-	-
101011 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101011 | 1 | 1 | 0	-	-	-	1.67%	44.41%	-	-	8.55%	6.34%	16.50%	22.53%	-	-
101011 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101011 | 1 | 2 | 0	-	-	-	3.13%	-	-	-	15.75%	10.23%	29.89%	41.01%	-	-
101011 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101100 | 0 | 1 | 0	-	5.75%	12.27%	0.85%	20.37%	20.69%	16.11%	3.93%	2.19%	7.50%	10.35%	-	-
101100 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101100 | 0 | 2 | 0	-	8.46%	-	1.23%	-	30.15%	24.04%	5.81%	3.74%	11.23%	15.35%	-	-
101100 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101100 | 1 | 1 | 0	-	-	-	1.74%	45.19%	-	-	8.54%	5.22%	16.53%	22.78%	-	-
101100 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101100 | 1 | 2 | 0	-	-	-	3.62%	-	-	-	15.69%	9.29%	30.00%	41.39%	-	-
101100 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101101 | 0 | 1 | 0	-	5.54%	11.70%	1.03%	20.60%	20.39%	17.05%	3.94%	1.82%	7.57%	10.37%	-	-
101101 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101101 | 0 | 2 | 0	-	8.18%	-	1.48%	-	30.31%	25.14%	5.77%	2.76%	11.11%	15.26%	-	-
101101 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101101 | 1 | 1 | 0	-	-	-	2.23%	45.47%	-	-	8.62%	4.50%	16.52%	22.66%	-	-
101101 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101101 | 1 | 2 | 0	-	-	-	3.94%	-	-	-	15.89%	7.43%	30.62%	42.13%	-	-
101101 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101110 | 0 | 1 | 0	-	5.47%	11.13%	1.14%	20.26%	20.69%	18.09%	3.90%	1.51%	7.54%	10.28%	-	-
101110 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101110 | 0 | 2 | 0	-	8.07%	-	1.72%	-	30.30%	26.21%	5.62%	2.07%	10.96%	15.06%	-	-
101110 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101110 | 1 | 1 | 0	-	-	-	2.45%	46.11%	-	-	8.74%	2.81%	16.82%	23.07%	-	-
101110 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101110 | 1 | 2 | 0	-	-	-	4.87%	-	-	-	15.84%	5.96%	31.00%	42.34%	-	-
101110 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101111 | 0 | 1 | 0	-	5.42%	10.27%	1.28%	20.49%	20.48%	19.29%	3.89%	0.99%	7.61%	10.27%	-	-
101111 | 0 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101111 | 0 | 2 | 0	-	7.96%	-	1.79%	-	29.80%	27.64%	5.57%	1.69%	10.76%	14.80%	-	-
101111 | 0 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101111 | 1 | 1 | 0	-	-	-	3.06%	45.99%	-	-	8.59%	2.59%	16.88%	22.89%	-	-
101111 | 1 | 1 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
101111 | 1 | 2 | 0	-	-	-	5.01%	-	-	-	15.92%	4.80%	31.63%	42.64%	-	-
101111 | 1 | 2 | 1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-

Table 17: *The classification table, part 6 of 8. The mask is arranged as:* $2^{\mathrm{nd}} - 7^{\mathrm{th}}$ *most significant bit of modulus* | 2^{nd} *least significant bit of modulus* | *modulus mod 3* | *modulus length mod 2.*

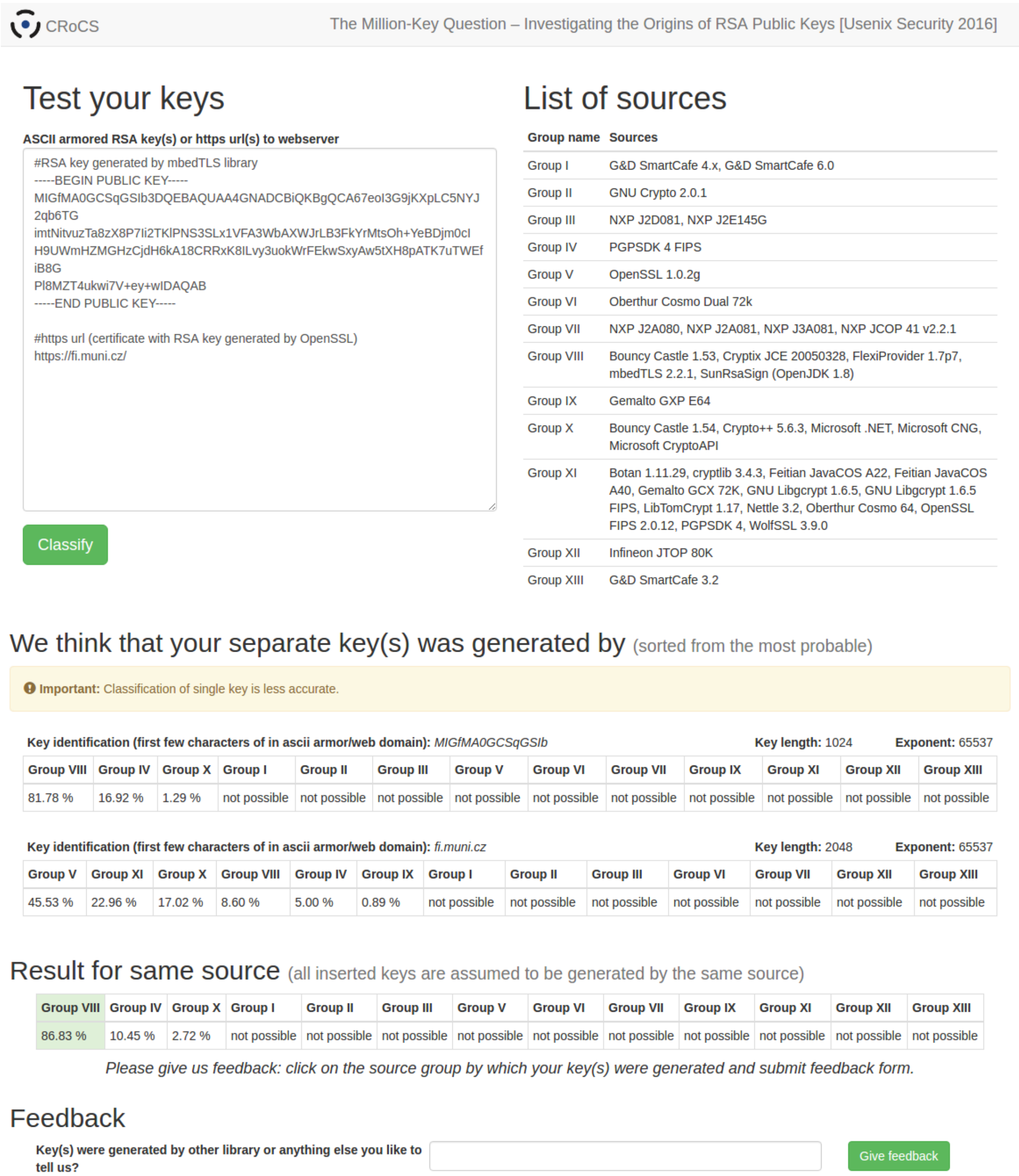
Mask value	Group of sources												
	I	II	III	IV	V	VI	VII	VIII	IX	X	XI	XII	XIII
110000∣0∣1∣0	-	5.54%	9.00%	1.48%	20.84%	20.09%	20.32%	3.88%	0.95%	7.55%	10.36%	-	-
110000∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110000∣0∣2∣0	-	7.99%	-	2.09%	-	29.38%	28.60%	5.50%	1.23%	10.67%	14.55%	-	-
110000∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110000∣1∣1∣0	-	-	-	3.53%	45.88%	-	-	8.60%	2.36%	16.75%	22.88%	-	-
110000∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110000∣1∣2∣0	-	-	-	6.48%	-	-	-	15.83%	3.61%	31.26%	42.82%	-	-
110000∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110001∣0∣1∣0	-	5.50%	8.13%	1.71%	21.08%	21.10%	19.57%	3.93%	0.57%	7.92%	10.49%	-	-
110001∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110001∣0∣2∣0	-	7.93%	-	2.53%	-	30.09%	27.22%	5.52%	1.01%	10.87%	14.83%	-	-
110001∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110001∣1∣1∣0	-	-	-	3.68%	46.07%	-	-	8.74%	1.67%	16.90%	22.94%	-	-
110001∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110001∣1∣2∣0	-	-	-	7.44%	-	-	-	15.82%	3.18%	31.42%	42.14%	-	-
110001∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110010∣0∣1∣0	-	5.73%	7.13%	1.99%	21.51%	21.97%	18.36%	4.07%	0.57%	7.89%	10.78%	-	-
110010∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110010∣0∣2∣0	-	8.28%	-	3.11%	-	30.43%	25.68%	5.64%	0.57%	11.20%	15.08%	-	-
110010∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110010∣1∣1∣0	-	-	-	4.30%	46.18%	-	-	8.57%	1.17%	16.90%	22.89%	-	-
110010∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110010∣1∣2∣0	-	-	-	8.20%	-	-	-	16.07%	1.43%	31.60%	42.71%	-	-
110010∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110011∣0∣1∣0	-	5.76%	6.21%	2.41%	22.66%	22.50%	16.74%	4.17%	0.31%	8.14%	11.10%	-	-
110011∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110011∣0∣2∣0	-	8.18%	-	3.73%	-	31.00%	23.37%	5.89%	0.63%	11.59%	15.61%	-	-
110011∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110011∣1∣1∣0	-	-	-	5.00%	45.53%	-	-	8.60%	0.89%	17.02%	22.96%	-	-
110011∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110011∣1∣2∣0	-	-	-	9.30%	-	-	-	15.71%	1.12%	31.55%	42.33%	-	-
110011∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110100∣0∣1∣0	-	5.99%	4.87%	3.08%	23.07%	23.85%	14.48%	4.33%	0.42%	8.49%	11.42%	-	-
110100∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110100∣0∣2∣0	-	8.49%	-	4.21%	-	32.29%	20.61%	6.00%	0.40%	11.90%	16.10%	-	-
110100∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110100∣1∣1∣0	-	-	-	5.78%	45.74%	-	-	8.54%	0.53%	16.79%	22.62%	-	-
110100∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110100∣1∣2∣0	-	-	-	10.99%	-	-	-	15.64%	1.20%	30.70%	41.47%	-	-
110100∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110101∣0∣1∣0	-	6.33%	3.07%	3.78%	24.12%	25.63%	11.78%	4.43%	0.18%	8.76%	11.92%	-	-
110101∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110101∣0∣2∣0	-	8.78%	-	5.39%	-	33.27%	16.83%	6.33%	0.20%	12.33%	16.86%	-	-
110101∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110101∣1∣1∣0	-	-	-	7.00%	45.66%	-	-	8.26%	0.37%	16.58%	22.13%	-	-
110101∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110101∣1∣2∣0	-	-	-	13.11%	-	-	-	15.46%	0.74%	29.86%	40.82%	-	-
110101∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110110∣0∣1∣0	-	6.68%	0.78%	4.48%	25.92%	26.71%	8.38%	4.75%	0.19%	9.45%	12.66%	-	-
110110∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110110∣0∣2∣0	-	8.98%	-	6.33%	-	35.67%	11.56%	6.63%	0.21%	13.17%	17.46%	-	-
110110∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110110∣1∣1∣0	-	-	-	8.09%	45.04%	-	-	8.19%	0.30%	16.45%	21.93%	-	-
110110∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110110∣1∣2∣0	-	-	-	14.76%	-	-	-	15.13%	0.53%	29.87%	39.70%	-	-
110110∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110111∣0∣1∣0	-	7.07%	-	5.81%	27.48%	27.38%	3.43%	5.08%	0.05%	10.15%	13.56%	-	-
110111∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110111∣0∣2∣0	-	9.73%	-	8.28%	-	37.66%	4.62%	7.16%	0.18%	13.82%	18.55%	-	-
110111∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110111∣1∣1∣0	-	-	-	9.62%	43.99%	-	-	8.36%	0.04%	16.30%	21.69%	-	-
110111∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
110111∣1∣2∣0	-	-	-	16.62%	-	-	-	14.86%	0.30%	29.04%	39.19%	-	-
110111∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-

Table 18: *The classification table, part 7 of 8. The mask is arranged as: $2^{\mathrm{nd}}-7^{\mathrm{th}}$ most significant bit of modulus ∣ 2^{nd} least significant bit of modulus ∣ modulus mod 3 ∣ modulus length mod 2.*

Mask value	Group of sources												
	I	II	III	IV	V	VI	VII	VIII	IX	X	XI	XII	XIII
111000∣0∣1∣0	-	7.43%	-	7.40%	28.35%	27.46%	-	5.23%	0.03%	10.35%	13.75%	-	-
111000∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111000∣0∣2∣0	-	10.20%	-	9.86%	-	38.60%	-	7.36%	0.04%	14.58%	19.35%	-	-
111000∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111000∣1∣1∣0	-	-	-	11.07%	43.45%	-	-	8.06%	0.14%	16.05%	21.23%	-	-
111000∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111000∣1∣2∣0	-	-	-	19.99%	-	-	-	14.24%	0.16%	28.29%	37.31%	-	-
111000∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111001∣0∣1∣0	-	7.08%	-	8.77%	27.57%	27.98%	-	5.08%	0.03%	10.16%	13.32%	-	-
111001∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111001∣0∣2∣0	-	9.80%	-	12.01%	-	39.05%	-	6.95%	-	13.90%	18.30%	-	-
111001∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111001∣1∣1∣0	-	-	-	13.54%	42.68%	-	-	7.65%	0.10%	15.53%	20.50%	-	-
111001∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111001∣1∣2∣0	-	-	-	23.17%	-	-	-	13.63%	-	27.19%	36.01%	-	-
111001∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111010∣0∣1∣0	-	6.99%	-	10.68%	27.00%	27.46%	-	4.92%	-	10.05%	12.90%	-	-
111010∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111010∣0∣2∣0	-	9.85%	-	14.63%	-	37.31%	-	6.83%	-	13.55%	17.82%	-	-
111010∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111010∣1∣1∣0	-	-	-	16.31%	40.94%	-	-	7.74%	0.06%	15.13%	19.82%	-	-
111010∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111010∣1∣2∣0	-	-	-	28.26%	-	-	-	12.81%	0.10%	25.27%	33.56%	-	-
111010∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111011∣0∣1∣0	-	6.56%	-	13.22%	26.15%	27.40%	-	4.76%	-	9.67%	12.24%	-	-
111011∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111011∣0∣2∣0	-	9.14%	-	18.52%	-	35.54%	-	6.71%	-	13.13%	16.97%	-	-
111011∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111011∣1∣1∣0	-	-	-	20.42%	39.07%	-	-	7.16%	-	14.67%	18.68%	-	-
111011∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111011∣1∣2∣0	-	-	-	32.15%	-	-	-	12.19%	-	24.30%	31.36%	-	-
111011∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111100∣0∣1∣0	-	6.49%	-	17.35%	24.97%	26.13%	-	4.50%	0.06%	9.15%	11.35%	-	-
111100∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111100∣0∣2∣0	-	8.78%	-	22.88%	-	34.60%	-	5.96%	-	12.47%	15.32%	-	-
111100∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111100∣1∣1∣0	-	-	-	25.09%	37.07%	-	-	7.01%	-	13.57%	17.27%	-	-
111100∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111100∣1∣2∣0	-	-	-	39.62%	-	-	-	10.74%	-	22.20%	27.45%	-	-
111100∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111101∣0∣1∣0	-	5.75%	-	23.37%	23.70%	24.10%	-	4.30%	-	8.46%	10.32%	-	-
111101∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111101∣0∣2∣0	-	7.70%	-	32.50%	-	29.63%	-	5.69%	-	11.01%	13.47%	-	-
111101∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111101∣1∣1∣0	-	-	-	33.93%	33.00%	-	-	6.14%	-	11.96%	14.97%	-	-
111101∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111101∣1∣2∣0	-	-	-	50.38%	-	-	-	9.31%	-	18.40%	21.91%	-	-
111101∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111110∣0∣1∣0	-	4.96%	-	34.36%	20.38%	20.20%	-	3.91%	-	7.51%	8.68%	-	-
111110∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111110∣0∣2∣0	-	6.36%	-	42.75%	-	26.30%	-	4.81%	-	9.21%	10.58%	-	-
111110∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111110∣1∣1∣0	-	-	-	47.36%	26.68%	-	-	4.91%	-	9.81%	11.25%	-	-
111110∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111110∣1∣2∣0	-	-	-	63.60%	-	-	-	6.71%	-	13.54%	16.15%	-	-
111110∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111111∣0∣1∣0	-	2.48%	-	63.73%	11.57%	10.77%	-	2.09%	-	4.33%	5.05%	-	-
111111∣0∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111111∣0∣2∣0	-	3.43%	-	71.63%	-	12.04%	-	2.43%	-	4.93%	5.55%	-	-
111111∣0∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111111∣1∣1∣0	-	-	-	73.23%	13.71%	-	-	2.51%	-	4.94%	5.60%	-	-
111111∣1∣1∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-
111111∣1∣2∣0	-	-	-	85.45%	-	-	-	2.75%	-	5.53%	6.27%	-	-
111111∣1∣2∣1	-	-	-	100.00%	-	-	-	-	-	-	-	-	-

Table 19: *The classification table, part 8 of 8. The mask is arranged as:* $2^{\text{nd}} - 7^{\text{th}}$ *most significant bit of modulus* ∣ 2^{nd} *least significant bit of modulus* ∣ *modulus mod 3* ∣ *modulus length mod 2.*

C Online classification tool



CRoCS

The Million-Key Question – Investigating the Origins of RSA Public Keys [Usenix Security 2016]

Test your keys

ASCII armored RSA key(s) or https url(s) to webserver

```
#RSA key generated by mbedTLS library
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCA67eoI3G9jKXpLC5NYJ
2qb6TG
imtNitvuzTa8zX8P7Ii2TKlPNS3SLx1VFA3WbAXWJrLB3FkYrMtsOh+YeBDjm0cI
H9UWmHZMGHzCjdH6kA18CRRxK8ILvy3uokWrFEkwSxyAw5tXH8pATK7uTWEf
iB8G
Pl8MZT4ukwi7V+ey+wIDAQAB
-----END PUBLIC KEY-----

#https url (certificate with RSA key generated by OpenSSL)
https://fi.muni.cz/
```

Classify

List of sources

Group name	Sources
Group I	G&D SmartCafe 4.x, G&D SmartCafe 6.0
Group II	GNU Crypto 2.0.1
Group III	NXP J2D081, NXP J2E145G
Group IV	PGPSDK 4 FIPS
Group V	OpenSSL 1.0.2g
Group VI	Oberthur Cosmo Dual 72k
Group VII	NXP J2A080, NXP J2A081, NXP J3A081, NXP JCOP 41 v2.2.1
Group VIII	Bouncy Castle 1.53, Cryptix JCE 20050328, FlexiProvider 1.7p7, mbedTLS 2.2.1, SunRsaSign (OpenJDK 1.8)
Group IX	Gemalto GXP E64
Group X	Bouncy Castle 1.54, Crypto++ 5.6.3, Microsoft .NET, Microsoft CNG, Microsoft CryptoAPI
Group XI	Botan 1.11.29, cryptlib 3.4.3, Feitian JavaCOS A22, Feitian JavaCOS A40, Gemalto GCX 72K, GNU Libgcrypt 1.6.5, GNU Libgcrypt 1.6.5 FIPS, LibTomCrypt 1.17, Nettle 3.2, Oberthur Cosmo 64, OpenSSL FIPS 2.0.12, PGPSDK 4, WolfSSL 3.9.0
Group XII	Infineon JTOP 80K
Group XIII	G&D SmartCafe 3.2

We think that your separate key(s) was generated by (sorted from the most probable)

Important: Classification of single key is less accurate.

Key identification (first few characters of in ascii armor/web domain): *MIGfMA0GCSqGSIb* **Key length:** 1024 **Exponent:** 65537

Group VIII	Group IV	Group X	Group I	Group II	Group III	Group V	Group VI	Group VII	Group IX	Group XI	Group XII	Group XIII
81.78 %	16.92 %	1.29 %	not possible	not possible	not possible	not possible	not possible	not possible	not possible	not possible	not possible	not possible

Key identification (first few characters of in ascii armor/web domain): *fi.muni.cz* **Key length:** 2048 **Exponent:** 65537

Group V	Group XI	Group X	Group VIII	Group IV	Group IX	Group I	Group II	Group III	Group VI	Group VII	Group XII	Group XIII
45.53 %	22.96 %	17.02 %	8.60 %	5.00 %	0.89 %	not possible	not possible	not possible	not possible	not possible	not possible	not possible

Result for same source (all inserted keys are assumed to be generated by the same source)

Group VIII	Group IV	Group X	Group I	Group II	Group III	Group V	Group VI	Group VII	Group IX	Group XI	Group XII	Group XIII
86.83 %	10.45 %	2.72 %	not possible	not possible	not possible	not possible	not possible	not possible	not possible	not possible	not possible	not possible

Please give us feedback: click on the source group by which your key(s) were generated and submit feedback form.

Feedback

Key(s) were generated by other library or anything else you like to tell us?

Give feedback

Figure 30: *Screenshot of our online classification tool, available at* `http://crcs.cz/rsapp`. *The website supports ASCII armored RSA keys and retrieval of RSA keys found in TLS certificate of supplied domain name. The keys are classified individually and as a group assumed to be generated by the same source.*