The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli

Matus Nemec
Masaryk University,
Ca' Foscari University of Venice
mnemec@mail.muni.cz

Marek Sys*
Masaryk University
syso@fi.muni.cz

Petr Svenda
Masaryk University
svenda@fi.muni.cz

Dusan Klinec
EnigmaBridge, Masaryk University
dusan@enigmabridge.com

Vashek Matyas
Masaryk University
matyas@fi.muni.cz

ABSTRACT

We report on our discovery of an algorithmic flaw in the construction of primes for RSA key generation in a widely-used library of a major manufacturer of cryptographic hardware. The primes generated by the library suffer from a significant loss of entropy. We propose a practical factorization method for various key lengths including 1024 and 2048 bits. Our method requires no additional information except for the value of the public modulus and does *not* depend on a weak or a faulty random number generator. We devised an extension of Coppersmith's factorization attack utilizing an alternative form of the primes in question. The library in question is found in NIST FIPS 140-2 and CC EAL 5+ certified devices used for a wide range of real-world applications, including identity cards, passports, Trusted Platform Modules, PGP and tokens for authentication or software signing. As the relevant library code was introduced in 2012 at the latest (and probably earlier), the impacted devices are now widespread. Tens of thousands of such keys were directly identified, many with significant impacts, especially for electronic identity documents, software signing, Trusted Computing and PGP. We estimate the number of affected devices to be in the order of at least tens of millions.

The worst cases for the factorization of 1024 and 2048-bit keys are less than 3 CPU-months and 100 CPU-years on single core of common recent CPUs, respectively, while the expected time is half of that of the worst case. The attack can be parallelized on multiple CPUs. Worse still, all susceptible keys contain a strong fingerprint that is verifiable in microseconds on an ordinary laptop – meaning that all vulnerable keys can be quickly identified, even in very large datasets.

KEYWORDS

RSA, factorization, smartcard, Coppersmith's algorithm

*M. Sys and M. Nemec contributed equally.

CCS'17, October 30–November 3, 2017, Dallas, TX, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4946-8/17/10…$15.00
DOI: http://dx.doi.org/10.1145/3133956.3133969

1 INTRODUCTION

RSA [69] is a widespread algorithm for asymmetric cryptography used for digital signatures and message encryption. RSA security is based on the integer factorization problem, which is believed to be computationally infeasible or at least extremely difficult for sufficiently large security parameters – the size of the private primes and the resulting public modulus N. As of 2017, the most common length of the modulus N is 2048 bits, with shorter key lengths such as 1024 bits still used in practice (although not recommend anymore) and longer lengths like 4096 bits becoming increasingly common. As the private part of the key is a very sensitive item, a user may use secure hardware such as a cryptographic smartcard to securely store and use the private key value.

Successful attacks against RSA based on integer factorization (finding the private primes p and q from the public modulus N) enable the attacker to impersonate the key owner and decrypt private messages. The keys used by secure hardware are of special interest due to the generally higher value of the information protected – e.g., securing payment transactions.

RSA requires two large random primes p and q, that can be obtained by generating a random candidate number (usually with half of the bits of N) and then testing it for primality. If the candidate is found to be composite, the process is repeated with a different candidate number.

However, there are at least three reasons to *construct* a candidate number from several smaller (randomly) generated components instead of generating it randomly: 1) an improved resistance against certain factorization methods, such as Pollard's $p - 1$ method [65]; 2) certification requirements such as the NIST FIPS 140-2 standard, which mandates that for all primes p, the values of $p - 1$ and $p + 1$ have at least one large (101-bit or larger) factor each; and 3) speedup of keypair generation, since testing random candidate values for primality is time consuming, especially on restricted devices like smartcards.

Yet, constructed primes may bring new problems as demonstrated in our work. In the past, practical attacks against RSA exploited the use of insecurely short key lengths susceptible to factorization via NFS [67] (e.g., 512-bit, still found on the Internet [38]); faulty or weak random number generators producing partially predictable primes, as in the electronic IDs of Taiwanese citizens [9]; software bugs causing primes to be generated from an insufficiently large space, as in the Debian RNG flaw [7]; or seeding with insufficient entropy, leading to multiple keypairs sharing a prime [38]. The knowledge or recovery of all bits of a private key is not always required for a successful attack thanks to the powerful technique proposed by Coppersmith [22]. If at least one half of the bits of one of the primes is known, the remaining bits can be computationally recovered. Then, even otherwise secure designs can be attacked by various side-channel and implementation-based attacks or by introducing faults into the computation.

Only on very rare occasions is an attacker potentially able to recover the private primes of a chosen key of a seemingly sufficient bit length, without physical access to the target device or a large amount of side-channel information. One notable attack that comes close is a simple GCD computation [10], which can quickly factorize a collection of moduli, but only if they happen to share a common prime, making the attack less likely to succeed on a single targeted keypair. The cause of such vulnerability is typically insufficient entropy during the keypair generation, as demonstrated for a large number of TLS and SSH keys [36, 38], therefore requiring multiple public keys to be created with the same malfunctioning implementation of a random number generator.

We present our attack against keys generated in cryptographic smartcards of *Infineon Technologies AG* (further denoted as *Manufacturer*), and our attack is *not* based on any weakness in a random bit generator or any additional side-channel information. Instead, the attack utilizes the specific structure of the primes as generated by *Manufacturer*'s on-chip cryptographic library (further denoted as *RSALib*[1]). We had access neither to the *RSALib*'s source code nor to the object code (since it is stored only in the secure on-chip memory and is not extractable), and the whole analysis was performed solely using RSA keys generated and exported from the *Manufacturer*'s cards and tokens.
In short, the paper has the following contributions:

1. Recovery of the internal structure of the primes: We identify the structure of RSA primes as produced by a black-box cryptographic library by a manufacturer of widely used cryptographic smartcards. The structure was recovered solely from our observations of statistical properties of large number of private keys generated in accordance with the specification of the product.

2. Practical factorization: We propose and implement a technique for the factorization of such RSA keys, with lengths including 1024 and 2048 bits, using our derivation of the methods by Coppersmith and Howgrave-Graham.

3. Fast detection algorithm: We design a very fast algorithm to verify whether a particular key originates from the inspected library based on the properties of the public modulus. The implementation was released[2] to allow users to check their own keys.

4. Analysis of impacted domains: We analyze multiple usage domains (TLS, PGP, eID, authentication tokens, software signing, etc.) for the prevalence of vulnerable keys and discuss the impact of key factorization.

The specific structure of the primes as generated by *RSALib* (most likely introduced to speed up prime generation) allows us to quickly identify keys generated by the library using only the public modulus (regardless of the length of the key) and to practically factorize RSA keys with various key lengths up to 2048 bits. The factorization method uses knowledge of the specific structure of such primes to apply our derivation of Coppersmith's method. Furthermore, we devised an alternative representation of the primes in question to make the attack computationally feasible on consumer hardware.

The impact is significant due to *Manufacturer* being one of the top three secure integrated circuit (IC) producers. Furthermore, the weakness lies in an on-device software library; hence, it is not limited just to a particular range of physical devices. The weakness can be traced back to at least the year 2012, increasing the number of affected domains. We assessed the impact in a several important real-world usage scenarios and made some recommendations for mitigation.

The fingerprinting method is fast, requiring just microseconds to run on a modulus. We successfully used the fingerprinting technique on large datasets of certificates, such as those submitted to Certificate Transparency logs, collected in Internet-wide TLS scans and stored on public PGP keyservers. This led to a discovery of thousands of keys in the wild with primes of the form in question. Our method has negligible false negative and false positive rates (observed as zero), as guaranteed by the very rare properties.

Where datasets with public RSA keys were not available (e.g., Trusted Platform Modules or EMV payment cards), we collected sample keys from different devices to get an idea about the typical key lengths used for the domain and to estimate the prevalence of devices producing potentially vulnerable keys.

Although the *RSALib* is not automatically shipped with every chip, the developers are motivated to deploy it in order to benefit from ready-to-use higher-level functions (such as the *RsaKeyGen* method in question) and to get an implementation designed with protections against side-channel and fault induction attacks in mind. However, even for certification, the *RSALib* is provided only as object files, without the source code [20].

The rest of our paper is organized as follows: Section 2 is intended for readers with interest in the mathematical details of the discovered flaw and the proposed factorization and fingerprinting methods. The readers interested mainly in the practical impacts should focus on the specifics of the implementation of the factorization method covered in Section 3 and the survey of impacted usage domains and the analysis of vulnerable keys found in the wild, as provided in Section 4. The possible approaches to short- and long-term mitigation are discussed in Section 5. Related work and conclusions are provided in Sections 6 and 7, respectively.

2 FINGERPRINTING AND FACTORIZATION

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

(1) Select two distinct large primes[3] p and q.
(2) Compute $N = p * q$ and $\varphi(N) = (p-1) * (q-1)$.
(3) Choose a public exponent[4] $e < \varphi(N)$, e 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).

[1]Likely RSA v1.02.013 library and later revisions.
[2]Full details and a tool for the detection of vulnerable keys can be found at https://crocs.fi.muni.cz/papers/rsa_ccs17.
[3]Generated randomly, but possibly constructed to achieve certain required properties.
[4]Usually with a low Hamming weight for faster encryption.

A factorization attack attempts to obtain p and q from the knowledge of N. Such an attack is believed to be computationally infeasible for sufficiently long N. The factorization can be sped up if some additional information about the private exponent d or about the primes p or q is known by the attacker.

2.1 Format of the constructed primes

Our motivation for a deeper analysis of the keys produced by *Manufacturer*'s devices stemmed from the observation of interesting statistical properties extracted from a large number of keys as described in [73]. When compared to other implementations and theoretical expectations on distribution of prime numbers, the keys exhibited a non-uniform distribution of (p mod x) and (N mod x) for small primes x. In this work, we recovered the structure responsible for the properties. All RSA primes (as well as the moduli) generated by the *RSALib* have the following form:

$$p = k * M + (65537^a \bmod M). \tag{1}$$

The integers k, a are unknown, and RSA primes differ only in their values of a and k for keys of the same size. The integer M is known and equal to some primorial $M = P_n\#$ (the product of the first n successive primes $P_n\# = \prod_{i=1}^{n} P_i = 2 * 3 * \cdots * P_n$). The value of M is related to the key size, where the key size is a multiple of 32 bits for keys generated by the *RSALib*. The value $n = 39$ (i.e., $M = 2 * 3 * \cdots * 167$) is used to generate primes for an RSA key with a key size within the $[512, 960]$ interval. The values $n = 71, 126, 225$ are used for key sizes within intervals $[992, 1952]$, $[1984, 3936]$, $[3968, 4096]$.

The most important property of the keys is that the size of M is large and almost comparable to the size of the prime p (e.g., M has 219 bits for the 256-bit prime p used for 512-bit RSA keys). Since M is large, the sizes of k and a are small (e.g., k has $256 - 219 = 37$ bits and a has 62 bits for 512-bit RSA). Hence, the resulting RSA primes suffer from a significant loss of entropy (e.g., a prime used in 512-bit RSA has only 99 bits of entropy), and the pool from which primes are randomly generated is reduced (e.g., from 2^{256} to 2^{99} for 512-bit RSA).

The specific format of the primes has two main consequences:

1. Fingerprinting: The keys are fingerprinted based on the existence of a discrete logarithm $\log_{65537} N \bmod M$. While the size of M is large, the logarithm can be computed easily since M has small factors only. The keys generated by the *RSALib* can be identified with a negligible error and within microseconds.

2. Factorization: During the factorization of N, we are looking for values of a, k. A naïve approach would iterate through different values of a (treating the value of $65537^a \bmod M$ as the "known bits") and apply Coppersmith's algorithm to find the unknown k, but the number of attempts is infeasibly large, as shown in Table 1. We found an alternative representation of the primes in question using smaller M' values (divisors of M), leading to a feasible number of guesses of the value a'. The reduction of M is possible since the entropy loss is sufficiently high to have more than enough known bits for the application of Coppersmith's algorithm to lengths including 1024 and 2048 bits.

2.2 Fingerprinting

The public RSA modulus N is a product of two primes p, q. The *RSALib* generates primes of the described form (1). The moduli have the corresponding form:

$$N = \overbrace{(k * M + 65537^a \bmod M)}^{p} * \overbrace{(l * M + 65537^b \bmod M)}^{q}, \tag{2}$$

for $a, b, k, l \in \mathbb{Z}$. The previous identity implies

$$N \equiv 65537^{a+b} \equiv 65537^c \bmod M, \tag{3}$$

for some integer c. The public modulus N is generated by 65537 in the multiplicative group $\mathbb{Z}_M^*$. The existence of the discrete logarithm $c = \log_{65537} N \bmod M$ is used as the fingerprint of the public modulus N generated by the *RSALib*.

2.2.1 Efficiency. Although the discrete logarithm problem is a hard problem in general, in our case, it can be computed within microseconds using the Pohlig-Hellman algorithm [64]. The algorithm can be used to efficiently compute a discrete logarithm for a group G, whose size $|G|$ is a smooth number (having only small factors). This is exactly our case with the group $G = [65537]$ (subgroup of $\mathbb{Z}_M^*$ generated by 65537). The size of G is a smooth number (e.g., $|G| = 2^4 * 3^4 * 5^2 * 7 * 11 * 13 * 17 * 23 * 29 * 37 * 41 * 53 * 83$ for 512-bit RSA) regardless of the key size. The smoothness of G is a direct consequence of the smoothness of M. Since M is smooth (M is a primorial, $M = 2 * 3 * 5 * \cdots * P_n$), the size of $\mathbb{Z}_M^*$ is even "smoother" ($|\mathbb{Z}_M^*| = \varphi(M)$). The size $|G|$ is a divisor of $|\mathbb{Z}_M^*|$ (from Lagrange's theorem), and it is therefore smooth as well.

2.2.2 False positives. The existence of the discrete logarithm serves as a very strong fingerprint of the keys. The reason is that while random primes/moduli modulo M cover the entire $\mathbb{Z}_M^*$, the *RSALib* generates primes/moduli from the group G – a tiny portion of the whole group. The sizes $|G|$ of the group G are listed in Table 1 in the Naïve brute force (BF) column. The size of $\mathbb{Z}_M^*$ is equal to $\varphi(M)$. For example, $|G| = 2^{62.09}$ while $|\mathbb{Z}_M^*| = \varphi(M) = 2^{215.98}$ for 512-bit RSA. The probability that a random 512-bit modulus N is an element of G is $2^{62-216} = 2^{-154}$. This probability is even smaller for larger keys. Hence, we can make the following conclusion with high confidence: an RSA key was generated by the *RSALib* if and only if the Pohlig-Hellman algorithm can find the discrete logarithm $\log_{65537} N \bmod M$. Our theoretical expectation was verified in practice (see Section 3.1) with no false positives found within a million of tested keys.

2.3 Factorization – attack principle

Our method is based on Coppersmith's algorithm, which was originally proposed to find small roots of univariate modular equations. In [22], Coppersmith showed how to use the algorithm to factorize RSA modulus N when high bits of a prime factor p (or q) are known. We slightly modified the method to perform the factorization with known p mod M (= $65537^a \bmod M$).

2.3.1 Coppersmith's algorithm. Coppersmith's algorithm is used as a parametrized black box in our approach. Parameters affect the success rate and running time of the algorithm. In order to optimize the entire factorization process, we optimized the parameters of Coppersmith's algorithm. We choose parameters so that the algorithm will certainly find unknown bits of the factor and so the computation time will be minimal. The fraction of known bits of the factor determines the optimal parameters (100% success rate, best speed) of the algorithm. Coppersmith's algorithm is slowest when using the required minimum of known bits (half of the bits of the factor). With more bits known, the running time of the algorithm decreases.

2.3.2 Naïve algorithm. For N of the form (2), we look for factor p or q. In order to find a prime factor (say, p), one has to find the integers k, a. A naïve algorithm would iterate over different options of $65537^a \bmod M$ and use Coppersmith's algorithm to attempt to find k. The prime p (q, respectively) is found for the correct guess of parameter a (b). The cost of the method is given by the number of guesses (ord) of a and the complexity of Coppersmith's algorithm. The term ord represents the multiplicative order of 65537 in the group $\mathbb{Z}_M^*$ ($ord = ord_M(65537)$) and can be computed simply using the technique described in Section 2.6. In practice, ord determines the running time of the entire factorization. The number of attempts is too high (see Table 1, *Naïve BF # attempts*) even for small key sizes. Decreasing the number of attempts is necessary to make the method practical.

2.3.3 Main idea. A crucial observation for further optimization is that the bit size of M is analogous to the number of known bits in Coppersmith's algorithm. It is sufficient to have just $log_2(N)/4$ bits of p for Coppersmith's algorithm [22]. In our case, the size of M is much larger than required ($log_2(M) > log_2(N)/4$). The main idea is to find a smaller M' with a smaller corresponding number of attempts $ord_{M'}(65537)$ such that the primes are still of the form (1), with M replaced by M' and a, k replaced by a', k'. The form of the primes p, q implies that the modulus N is of the form (2) and (3) also for M' – of course with new corresponding variables a', b', c', k', l'.

In order to optimize the naïve method, we are looking for M' such that:

(1) primes (p, q) are still of the form (1) – M' must be a divisor of M;
(2) Coppersmith's algorithm will find k' for correct guess of a' – enough bits must be known ($log_2(M') > log_2(N)/4$);
(3) overall time of the factorization will be minimal – number of attempts ($ord_{M'}(65537)$) and time per attempt (running time of Coppersmith's algorithm) should result in a minimal time.

For practical factorization, there is a trade-off between the number of attempts and the computational time per attempt as Coppersmith's algorithm runs faster when more bits are known (see Figure 2). In fact, we are looking for an optimal combination of value M' and parameters (m, t – for more details, see Section 2.7) of Coppersmith's algorithm. It should be noted that the search for value of M' is needed only once for each key size. The optimal parameters M', m, t along with N serve as inputs to our factorization Algorithm 1.

2.3.4 Results. The optimized values of M' for different key lengths were found along with parameters m, t using a local brute force search optimized by the results of a greedy heuristic. The size of the resulting M' is more than the bound $log_2(N)/4$ but is relatively close to it. The resulting order (Table 1, *Optimized BF # attempts*) is small enough for the factorization of 512, 1024 and 2048-bit RSA to be practically feasible. Figure 1 summarizes the factorization complexity and relevant parameters for all key lengths between 512 and 4096 bits with 32-bit steps. The search space of a' can be trivially partitioned and parallelized on multiple CPUs. We verified the actual performance of the proposed factorization method on multiple randomly selected public keys.

```
Input   : $N, M', m, t$
Output : $p$ – factor of $N$
$c' \leftarrow \log_{65537} N \bmod M'$        ▷ Use Pohlig–Hellman alg;
$ord' \leftarrow ord_{M'}(65537)$        ▷ See Section 2.6 for method;
forall $a' \in \left[\frac{c'}{2}, \frac{c'+ord'}{2}\right]$ do
    $f(x) \leftarrow x + (M'^{-1} \bmod N) * (65537^{a'} \bmod M') \pmod N$;
    $(\beta, X) \leftarrow (0.5, 2 * N^{\beta}/M')$        ▷ Setting parameters;
    $k' \leftarrow Coppersmith(f(x), N, \beta, m, t, X)$;
    $p \leftarrow k' * M' + (65537^{a'} \bmod M')$ ▷ Candidate for a factor;
    if $N \bmod p = 0$ then
        return $p$
    end
end
```
Algorithm 1: The factorization algorithm for RSA public keys N generated by the *RSALib*. The input of the algorithm is a modulus N of the form (1) with M' as a product of small primes and optimized parameters m, t for Coppersmith's method.

2.4 Coppersmith's algorithm in detail

There are various attacks on RSA based on Coppersmith's algorithm (for a nice overview, see [57]). The algorithm is typically used in scenarios where we know partial information about the private key (or message) and we want to compute the rest. The given problem is solved in the three steps:

$$problem \quad \rightarrow \quad f(x) \equiv 0 \bmod p \quad \rightarrow \quad g(x) = 0 \quad \rightarrow \quad x_0$$

First, we transform the given *problem* to the modular polynomial equation $f(x) \equiv 0 \bmod p$ with an unknown p (divisor of some known N) and the small root x_0 ($f(x_0) \equiv 0 \bmod p$) we are looking for. The root x_0 should be smaller than some sufficiently small constant X (i.e., $|x_0| < X$). The polynomial $f(x) \in \mathbb{Z}[x]$ should be constructed so that the root x_0 solves the *problem*. In the second step, Coppersmith's algorithm eliminates the unknown p by transforming the modular equation to the equation $g(x) = 0$ over the integers that have the same roots (i.e., x_0 is a root of $g(x)$). In the third step, all roots x_0 of the integer polynomial $g(x)$ are found easily by standard methods (e.g., the Berlekamp-Zassenhaus algorithm [8, 21]).

The polynomial $g(x)$ is constructed in Coppersmith's algorithm as a linear combination $g(x) = \sum_i a_i * f_i(x), a_i \in \mathbb{Z}$ of some polynomials $f_i(x)$ derived from $f(x)$. The polynomials f_i are chosen so that $f_i(x)$ and $f(x)$ have the same roots modulo p. This implies that $g(x)$ and $f(x)$ share the same roots (with some additional roots) modulo p as well. The main idea of Coppersmith's algorithm is to find $g(x) \in \mathbb{Z}[x]$ such that $|g(x_0)| < p$, which means that the equivalence $g(x_0) \equiv 0 \bmod p$ also holds over the integers, i.e., $g(x_0) = 0$.

Key size	M	Size of M	Size of M'	Naïve BF # attempts ($ord_M(65537)/2$)	Our BF # attempts ($ord_{M'}(65537)/2$)	Time per attempt	Worst case
512 b	$P_{39}\# = 167\#$	219.19 b	140.77 b	$2^{61.09}$	$2^{19.20}$	11.6 ms	1.93 CPU hours
1024 b	$P_{71}\# = 353\#$	474.92 b	285.19 b	$2^{133.73}$	$2^{29.04}$	15.2 ms	97.1 CPU days
2048 b	$P_{126}\# = 701\#$	970.96 b	552.50 b	$2^{254.78}$	$2^{34.29}$	212 ms	140.8 CPU years
3072 b	$P_{126}\# = 701\#$	970.96 b	783.62 b	$2^{254.78}$	$2^{99.29}$	1159 sec	$2.84 * 10^{25}$ years
4096 b	$P_{225}\# = 1427\#$	1962.19 b	1098.42 b	$2^{433.69}$	$2^{55.05}$	1086 ms	$1.28 * 10^{9}$ years

Table 1: Overview of the used parameters (original M and optimized M') and performance of our factorization algorithms for commonly used key lengths. Time measurements for multiple attempts were taken on one core of an Intel Xeon E5-2650 v3 CPU clocked at 3.00 GHz, and the worst case time estimates are extrapolated from the orders and the average times required per attempt. The expected factorization time is half of the worst case time.

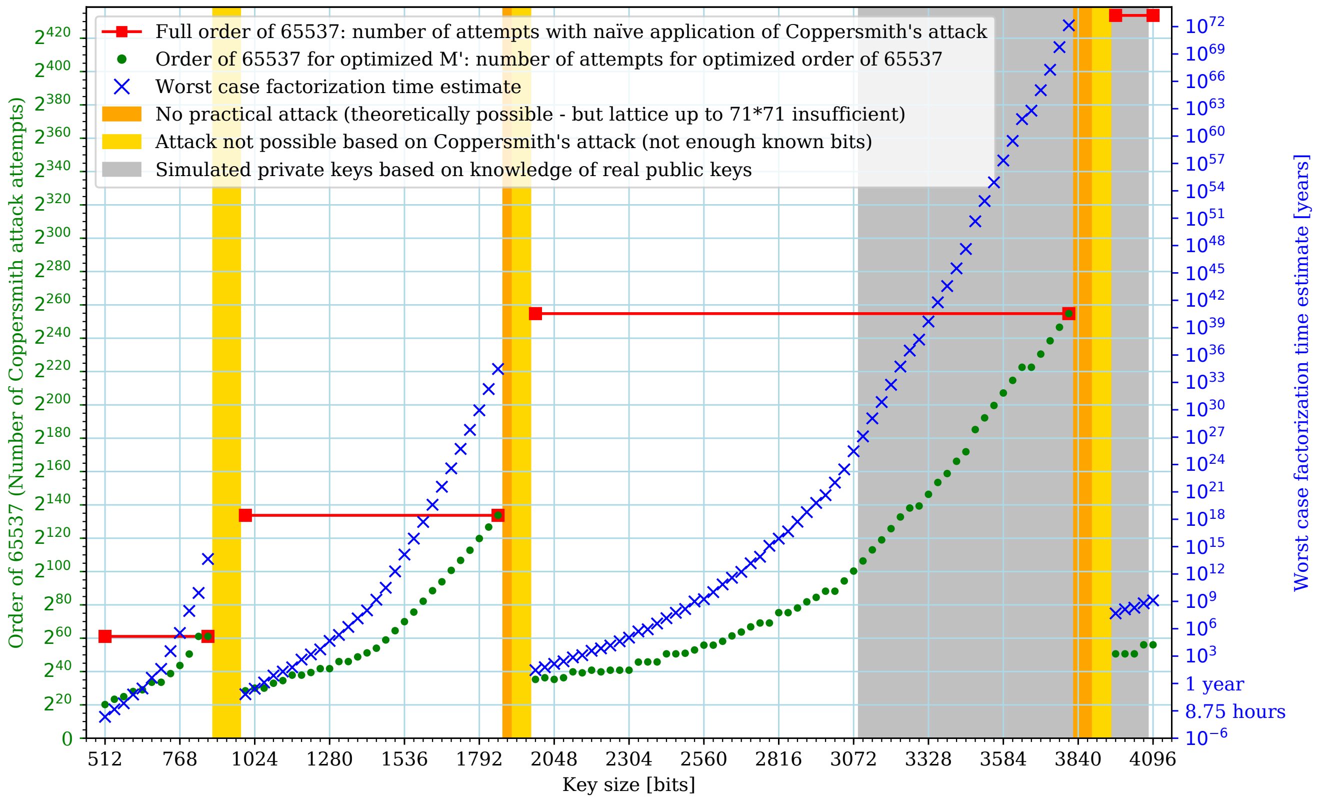

Figure 1: The complexity of the factorization of keys produced by the studied *RSALib* with different key lengths starting from 512 to 4096 bits in 32-bit steps (horizontal axis). The blue crosses show the worst case estimate for the time to factorize a key with the given length, with the vertical axis scale on the right side showing the estimated CPU effort on one core of an Intel Xeon E5-2650 v3 CPU clocked at 3.00 GHz. The red lines show the full order of the group. The green dots show the reduced order as achieved by our method. The yellow areas indicate the key lengths for which our method, which is based on Coppersmith's attack, is not applicable due to an insufficient number of known bits. The orange areas indicate the key lengths where the attack should be possible in practice; however, we were not successful in finding suitable parameters. The gray area shows the key lengths where only public keys were available to us; hence, we simulated the private keys for the computations backing the creation of the graph (since the structure of the keys can be recovered from either private or public keys, the simulation should be sufficient).

The polynomial $g(x)$ is found by the LLL algorithm [51] using the fact that the root x_0 is small.

The LLL algorithm reduces a lattice basis $b_0, \cdots, b_{n-1}$. The algorithm computes an alternative basis $b'_0, \cdots, b'_{n-1}$ of the lattice such that the vectors $b'_0, \cdots, b'_{n-1}$ are smaller than the vectors in the original basis. The LLL algorithm is typically used to find one sufficiently short vector of the lattice. The algorithm is applied to the matrix B, which consists of row vectors b_i. The result of the reduction is the matrix B' of short vectors b'_j, which are all constructed as linear (but with integer coefficients) combinations of basis vectors b_i. Coppersmith's algorithm utilizes the LLL algorithm in order to find the desired polynomial $g(x)$ with a small function value $g(x_0)$. The LLL is used to find an "equivalent" polynomial $g(xX)$ (x – a variable, X – a known constant) rather than $g(x)$. The LLL is used here to find the polynomial $g(xX)$ as a linear combination of polynomials $f_i(xX)$. The LLL algorithm is applied to the matrix B that consists of coefficient vectors of polynomials $f_i(xX)$ for $|x_0| < X$. The polynomial $g(x)$ is defined by the smallest vector b'_0 of the reduced basis as $g(x) = \sum_{i=0}^{n-1} b'_{0,i} x^i$ for $b'_0 = [b'_{0,0}, b'_{0,1}, \cdots, b'_{0,n-1}]$. A small norm $|b'_0| = \sqrt{\sum_{i=0}^{n-1} (b'_{0,i} X^i)^2}$ of the vector b'_0 implies small function value $|g(x_0)| = |\sum_{i=0}^{n-1} b'_{0,i} x_0^i|$ [57, Proof of Theorem 2].

2.5 Application of Coppersmith's algorithm

Our factorization is based on the SageMath implementation [75] of the Howgrave-Graham method [40] – a revisited version of Coppersmith's algorithm.

2.5.1 Howgrave-Graham method. In general, Coppersmith's algorithm and the Howgrave-Graham method use p^m instead of p, i.e., x_0 is root of polynomials $f_i(x)$ modulo p^m, and we are looking for $g(x)$ such that $|g(x_0)| < p^m$. The method uses the following set of polynomials $f_i(x)$ generated as:

$$f_i(x) = x^j N^i f^{m-i}(x) \quad i = 0, \cdots, m-1, \quad j = 0, \cdots, \delta - 1, \quad (4)$$

$$f_{i+m}(x) = x^i f^m(x) \qquad i = 0, \cdots, t-1, \qquad (5)$$

and parametrized by the degree δ of the original polynomial $f(x)$ ($\delta = 1$ in our case). The Coppersmith-Howgrave-Graham method is further parametrized by three parameters m, t, X (apart from $f(x), N$), defining the matrix B and influencing the running time. The parameters m, t define the number of polynomials $n = \delta * m + t$ and the dimension of the square matrix B. The third parameter X – the upper bound for the solutions we are looking for ($x_0 < X$) – determines the bit size of the entries of the matrix B. The running time of Coppersmith's algorithm is dominated by the time needed for the LLL reduction. The running time of the LLL reduction is given by the matrix B (the row dimension and the size of its entries) and is mostly determined by the matrix size n.

2.5.2 Application to (p mod M') known. The Howgrave-Graham method is able to find a sufficiently small solution x_0 of the equation $f(x) = 0 \bmod p$. In our case, the primes p, q are of the form $p = k' * M' + (65537^{a'} \bmod M')$, with the small k' being the only unknown variable of the equation. Hence, the polynomial $f(x)$ can be constructed as $f(x) = x * M' + (65537^{a'} \bmod M')$, since $f(x_0) = 0 \bmod p$ has a small root $x_0 = k'$. The method requires $f(x)$ to be monic (the leading coefficient is 1), but the form can be easily obtained [57] as:

$$f(x) = x + (M'^{-1} \bmod N) * (65537^{a'} \bmod M') \pmod{N}. \quad (6)$$

2.5.3 Setting the parameters X and β. The parameter β represents the upper bound for the ratio of the bit size of the factor p and the modulus N, i.e., $p < N^{\beta}$. Since the bit size of both primes p, q is half of the bit size of N, the value β is set to 0.5. The parameter X represents the upper bound for the solution x_0 of the modular polynomial equation. In our case, X represents an upper bound for the value of k' from the equation (1); hence, its value can be computed as $X = 2 * N^{0.5}/M'$.

2.6 Computing the order of a generator in $\mathbb{Z}^*_{M'}$

The order of the generator 65537 is used in our Algorithm 1 and also for the optimization of parameters (see the next section). The multiplicative order $ord' = ord_{M'}(65537)$ is the smallest non-zero integer such that $65537^{ord'} \equiv 1 \bmod M'$, which is equivalent to $65537^{ord'} \equiv 1 \bmod P_i$ for all prime divisors P_i of M'. Since M' is the product of different primes, the ord' can be computed as the least common multiple of the partial orders $ord_{P_i} = ord_{P_i}(65537)$ for primes divisors P_i of M':

$$ord' = lcm(ord_{P_1}, ord_{P_2}, \cdots) \text{ for } P_i | M'. \quad (7)$$

2.7 Optimization of the parameters M', m, t

The optimization of parameters is performed only once for all RSA keys of a given size. The parameters M', m, t affect the success rate and the running time of our method. We are looking for parameters M', m, t such that the success rate is 100% (k' is certainly found for a correct guess of a') and the overall time is minimal. The running time (the worst case)

$$Time = ord_{M'}(65537) * T(M', m, t)$$

of our method is determined by the number of guesses ($ord_{M'}(65537)$ for the parameter a') and the computation average running time T for one attempt (computation of k' using Coppersmith's algorithm). The running time of Coppersmith's algorithm is dominated by the LLL reduction so it is affected mostly by the size $n = m + t$ of the square matrix B and partially by the size of matrix elements given by the size of M'. The value M' affects both the number of attempts ($ord_{M'}(65537)$) and the time for one attempt ($T(M', m, t)$); hence, we optimize all parameters M', m, t together. During the optimization, we focus on decreasing the value $ord' = ord_{M'}(65537)$. The optimization process can be described as follows:

(1) Compute a set of candidates for M', each candidate with sufficiently small corresponding ord';
(2) For each candidate, find the optimal (100% success rate, minimal time per attempt) parameters m, t – only reasonably small parameters m, t are brute-forced ($t = m + 1$ and $m = 1, \cdots, 35$). For a given m, t, the Howgrave-Graham method is applied to the polynomial (6) for correct guess of a' (to compute success rate) and also for incorrect guesses of a' (to compute the average time per attempt);
(3) Return the best combination M', m, t with the minimal corresponding $Time$.

The best combination M', m, t was obtained with respect to the implementation [75] of the Howgrave-Graham method applied to

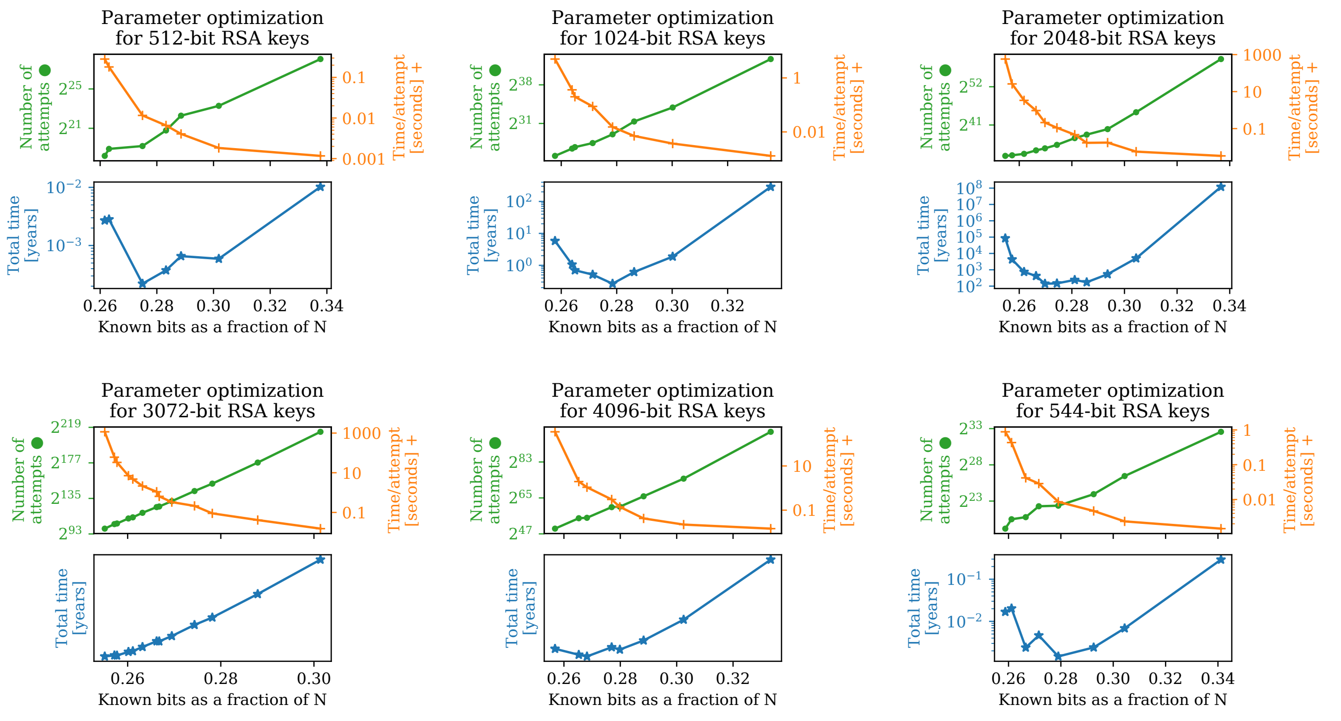

Figure 2: The trade-off between the number of attempts (green circles) and the time per attempt (orange crosses) as the function of the number of known bits (size of the optimized M'). We select the parameters corresponding to the minimal overall time of the factorization (blue stars). Values with the same lattice size perform an attempt in an approximately same time. The best number of attempts for each considered lattice size ($m + t$) is plotted. There can exist more than one local minimum for the total time (as seen for 544-bit RSA keys). Please notice the logarithmic scale of the vertical axis.

```
Input  : primorial $M$, $ord'$ – divisor of $ord_M(65537)$
Output : $M'$ of maximal size with $ord_{M'}(65537)|ord'$
$M' \leftarrow M$;
forall primes $P_i|M$ do
    $ord_{P_i} \leftarrow ord_{P_i}(65537)$;
    if $ord_{P_i} \nmid ord'$ then
        $M' \leftarrow M'/P_i$;
    end
end
return $M'$
```

Algorithm 2: The computation of the maximal divisor M' of the primorial M with $ord_{M'}(65537)|ord'$ for a given ord' (divisor of $ord_M(65537)$).

keys of a given size. We used a dataset of RSA keys of given sizes (512 to 4096 bits, by 32-bit increments) with known factorizations and having our special form (2). The approximate size of the optimized M' for various key lengths can be found in Figure 1. The most common key lengths used the following m, t values: $m = 5, t = 6$ for 512, $m = 4, t = 5$ for 1024, $m = 6, t = 7$ for 2048, $m = 25, t = 26$ for 3072, $m = 7, t = 8$ for 4096.

2.7.1 Optimizing M'. In order to preserve the format of the primes, we are looking for a divisor M' of M (see Section 2.3.3) that is a primorial $M = 2 * 3 * \cdots * P_n$. Divisor M' of M is selected as a candidate for an optimal M' (with the best m, t) if the value $ord_{M'}(65537)$ is sufficiently small but the size M' is large enough (Coppersmith's algorithm requires $log_2(M') > log_2(N)/4$).

Our aim is to perform a brute force search for M'. In order to speed up the search we are looking for the value $ord_{M'}(65537)$ rather than M'. Once $ord' = ord_{M'}(65537)$ is found, the maximal corresponding value M' can be computed easily. Although the search space for ord' is smaller than the space for M', the brute force search is still feasible only for smaller key sizes. Hence, we used a combination of two heuristics – greedy and local brute force search.

The general strategy is to maximize the size of M' and simultaneously minimize the corresponding order. The value M' for given key size was found in two steps:

- First, we used a greedy heuristic (with a "tail brute force phase") to find an "almost" optimal M', denoted by M'_{greedy} with the corresponding order ord'_{greedy}.
- Second, the value ord'_{greedy} was used to reduce the search space of a "local" brute force search for a better M'.

In both strategies, we used the simple Algorithm 2 that given ord' looks for the maximal M' (divisor of M) such that given ord' equals $ord_{M'}(65537)$. In some cases, no such M' exists, then Algorithm 2 finds M' such that the corresponding order $ord_{M'}(65537)$ is the maximal proper divisor of given ord'. Algorithm 2 is based on the formula (7). The algorithm eliminates from M only those prime divisors $P_i|M$ whose partial order $ord_{P_i}(65537)$ does not divide given ord'.

2.7.2 Greedy heuristic. In the greedy strategy, we try to minimize $ord_{M'}(65537)$ and simultaneously maximize the size of M' (to get $log_2(M') > log_2(N)/4$). The greedy heuristic is an iterated strategy with local optimal improvement performed in each iteration. In each iteration, we reduce (divide) ord' by some prime power divisor $p_j^{e_j}$ and compute the corresponding M' of maximal size using Algorithm 2. In the greedy choice, we select the most "valuable" prime power divisor $p_j^{e_j}$ of ord' that provides a large decrease in the order ord' at a cost of a small decrease in the size of M'. The divisor is chosen as the highest reward-at-cost value, defined as:

$$\frac{\Delta\text{size of } ord_{M'}}{\Delta\text{size of } M'} = \frac{\log_2(ord_{M'_{old}}) - \log_2(ord_{M'_{new}})}{\log_2(M'_{old}) - \log_2(M'_{new})}$$

for M'_{new} computed by Algorithm 2 with M'_{old}, $ord' = ord_{M'_{old}}/p_j^{e_j}$ as an input. The reward-at-cost represents the bit size reduction of the order at the cost of the bit size reduction of M'. The following example illustrates how our greedy heuristic works:

Example 2.1. The initial M'_{old} for RSA–512 is set to $M = P_{39}\# = 167\# = 2 * 3 * \cdots * 167$. The factorization of the initial order is: $ord' = ord_{M'_{old}} = 2^4 * 3^4 * 5^2 * 7 * 11 * 13 * 17 * 23 * 29 * 37 * 41 * 83$. There are 19 candidates $2^1, \cdots, 2^4, 3, \cdots, 3^4, 5, 5^2, 7, \cdots, 83$ for the most valuable prime power divisor $p_j^{e_j}$ of ord' in the first iteration. For the candidate $p_j^{e_j} = 83^1$, Algorithm 2 eliminates 167 from M'_{old} since $83^1|ord_{167} = 166$ and $83^1 \nmid ord'$. Algorithm 2 returns $M'_{new} = M'_{old}/167$ for the input values $M_{old}, ord' = ord_{M'_{old}}/83^1$. The reward-at-cost for 83^1 is computed as $\log_2 83/\log_2 167 = \frac{6.37}{7.38}$ for the reduction of the order by 6.37 bits and the reduction of M' by 7.38 bits. For the candidate 17^1, Algorithm 2 eliminates 103, 137 (i.e., $M'_{new} = M'_{old}/(103 * 137)$), since $17|ord_{103} = 51 = 17 * 3, 17|ord_{137} = 136 = 17 * 8$ and $17^1 \nmid ord_{M'_{old}}/17$. The reward-at-cost for 17^1 is computed as $\log_2 17/\log_2(103 * 137) = 4.08/13.78$, *etc*. The most valuable candidate in the first iteration is $p_j^{e_j} = 83^1$ with the highest reward-at-cost 0.8633.

In the second iteration, we start with $M'_{old} = M/167$ and $ord' = ord_{M'_{old}} = 2^4 * 3^4 * \cdots * 7 * 11 * 13 * 17 * 23 * 29 * 37 * 41$ and compute the new reward-at-cost for all 18 candidates $2^1, \cdots, 2^4$, $3, \cdots, 3^4, 5, 5^2, 7, \cdots, 41$. In the second iteration, the best candidate for divisor $p_j^{e_j}$ of new ord' with the highest reward-at-cost is $p_j^{e_j} = 53^1$, etc.

Throughout the iterations, the following best candidates for $p_i^{e_i}$ are found: $83^1, 53^1, 41^1, 29^1, 37^1, 23^1, 17^1, 3^2, 11^1$.

In the last iteration, the greedy heuristic computes M' that is too small ($\log_2(M') < \log_2(N)/4$), which finishes the computation. The resulting M' = `0x55eb8fbb4ca1e1879d77` from the previous iteration is computed by Algorithm 2 for $M' = M/83/53/\cdots/11$.

The greedy method returns an optimal M' for 512-bit RSA keys. The optimality was verified by brute force, testing all possible divisors of M with sufficiently small corresponding order.

2.7.3 Tail brute force. The greedy strategy can be improved for larger keys by brute force testing all divisors of $ord_{M'}$ that is found by the greedy heuristic. First, we execute the greedy strategy, that gives us the sequence of the values of $M'_0 > M'_1 > \cdots > M'_L$ from the iterations $0, 1, \cdots, L$. Then, we use brute force (testing all divisors) for $ord_{M'_i}$, starting with M'_{L-1} and continuing with $M'_{L-2}, \cdots$, limited by reasonable running time.

2.7.4 Local brute force. There are two ways to perform the brute force search for an optimized M' (divisor of M). We can search through divisors M' of M, or we can use an alternative search through all divisors ord' of $ord_M(65537)$ ($M'|M \implies ord_{M'}|ord_M$) and compute the corresponding M' from ord' using Algorithm 2. We use the second approach since the search space for ord' is significantly smaller than that for M'. For example, for 512-bit RSA keys, $M = P_{167}\#$ is product of 39 primes, i.e., there are 2^{39} different divisors of M, while there are only $5^2 * 3 * 2^9 \approx 2^{15}$ different divisors of $ord_M(65537) = 2^4 * 3^4 * 5^2 * 7 * 11 * 13 * 17 * 23 * 29 * 37 * 41 * 83$.

For smaller key sizes, it is possible to search through all divisors of the order, but for large key sizes (e.g., 4096-bit RSA), the brute force strategy is infeasible and needs to be optimized. We implemented an algorithm that recursively iterates through all divisors ord' of $ord_M(65537)$. Recursion allows us to optimize the search and to skip inappropriate candidates (small M', big $ord_{M'}(65537)$) for an optimal M'.

We use two approaches that recursively iterate through orders:

- Decreasingly – In this approach we start with the full order $ord' = ord_M(65537)$, and in each iteration, we divide $ord' = ord'/p_j$ by a prime divisor of current ord'. The branch of the recursion is stopped when M' is too small ($log_2(M') < log_2(N)/4$). This approach is suitable for key sizes with bit sizes of M' close to the lower bound $log_2(N)/4$ because only several primes p_j can be eliminated from ord' and most inappropriate candidates are skipped due to a small size of M'.
- Increasingly – We start with $ord' = 1$ and in each step multiply the order $ord' = ord' * p_j$ by some prime divisor p_j of $ord_M(65537)$. When ord' is too large, we stop the given branch of the recursion and skip the worst candidates. As an upper bound for ord', we use the value $ord_{greedy} * 2^5$. This approach is suitable for key sizes for which the bit size of M' is significantly bigger than $log_2(N)/4$ since most candidates are skipped due to the large value of ord'.

2.8 Guessing strategy

Our method can find the prime factor p for the correct guess x of a'. A simple incremental search $x = 0, 1, \cdots$ for a' would iterate through $ord_{M'}(65537)$ for different values of x in the worst case since

$$p = 65537^{a'} \bmod M'.$$

Denoting $ord' = ord_{M'}(65537)$, we are looking for $x \equiv a' \bmod ord'$.

Since both p, q are of the same form, our method can also find the factor q for $x \equiv b' \bmod ord'$. Hence, our method is looking simultaneously for p and q. This fact can be used to halve the time needed to find one of the factors p, q of N. In order to optimize the guessing strategy, we are looking for the smallest subset (interval) of $\mathbb{Z}_{ord'}$ that contains either a' or b'. We use the value c' obtained during the fingerprinting (a discrete logarithm of N) to describe the desired interval. The interval is of the following form:

$$I = \left[\frac{c'}{2}, \frac{c' + ord'}{2}\right].$$

It is easy to see that either a' or b' ($c' \equiv a' + b' \bmod ord'$) occur in the interval I and that the size of the I is the smallest possible.

3 PRACTICAL IMPLEMENTATION

We implemented the full attack in SageMath, based on an implementation [75] of the Howgrave-Graham method [40]. We used it to verify the applicability of the method on real keys generated on the vulnerable smartcards. It was also used to perform time measurements in order to optimize our parameters and evaluate the worst case running time, as captured by Figure 1 and Table 1.

3.1 Details and empirical evaluation

The fingerprint verification algorithm computes the discrete logarithm of a public modulus. We chose the primorial of 512-bit RSA as the modulus, since it applies to all key lengths. We recorded no false negatives in 3 million vulnerable keys generated by *RSALib*, since all of the keys have the sought structure. As expected, no false positives were recorded on 1 million non-affected keys generated by OpenSSL. We estimated the probability of a false positive on a single key as 2^{-154} in Section 2.2.2.

We practically verified the factorization method on multiple randomly selected 512 and 1024-bit keys. Since the complexity of factorization of a 2048-bit key could be approximately 100 CPU years, we did not select keys randomly. Instead, we generated keys on an affected smartcard and exported the private keys. The knowledge of the primes allows us to precisely compute the number of attempts required for the factorization as the distance of the initial guess $c'/2$ (Section 2.8) to a' or b' (whichever is closer). Out of 137,000 freshly generated keys, we selected 24 public keys with the least effort required (all keys with 2^{21} attempts or fewer) for factorization and ran the computation, each finishing within one week. We used the time measurements to verify the linear relationship of factorization time on the order and we checked that the worst case time estimate matches the slope of the line.

3.2 Possible improvements and limitations

The attack can be trivially parallelized on multiple computers. Each individual task is assigned a different subrange of the values a' that need to be guessed. The expected wall time of the attack can be decreased linearly with the number of CPUs (assuming that each task can execute the same number of attempts per a unit of time). However, the expected CPU time and the worst case CPU time remain unaffected.

The time of each attempt is dominated by lattice reduction. Our implementation uses the default implementation of LLL in SageMath (backed by the fpyLLL wrapper for fpLLL [27]). A more efficient implementation might speed up the process. However, we do not expect significant improvements.

In our opinion, the best improvement could be achieved by a better choice of polynomials in the phase of lattice construction. We follow the general advice for polynomial choice from [57]. More suitable lattice may exist for our specific problem.

Our algorithm for optimizing the running time utilizes a heuristic for finding an optimized value of the modulus M'. A better heuristic or a bruteforce search might find a modulus, where the generator has a lower order or could discover a better combination of the lattice size and M' value.

Despite an extensive search for better values within a significantly larger space (Section 2.7.4), we obtained only small improvements of the overall factorization time (halving the overall time at best in comparison to the greedy algorithm). We examined the trade-off between the number of attempts and the time per attempt, as captured by Figure 2 to understand the nature of the optimization process.

We did not explore implementations of lattice reduction backed by dedicated hardware or GPUs. Most key lengths are processed with a lattice of low dimensions, however, some improvements may be gained for lengths that require a large lattice [39]. In our experience, the memory used by one factorization was up to 300 MB. SageMath is an interpreted language, so the requirements of a hardware circuit might be different.

Finally, we cannot rule out that a fundamentally improved approach, which would utilize the properties of keys more efficiently, will be devised.

4 ANALYSIS OF IMPACTS

The discussion of impacts is far from straightforward. First, the prevalence of factorizable keys in a given usage domain is between very easy to very difficult to obtain. For example, the prevalence of fingerprinted keys used for TLS is easy to enumerate thanks to Internet-wide scans like Censys [28]. Obtaining large datasets of public keys for usage domains for devices expected to be more vulnerable (e.g., electronic passports) is usually significantly harder given the nature of secure hardware use.

Secondly, the actual damage caused by a factorized key varies significantly between and also within the usage domains. Finally, not all key lengths are actually factorizable, and the factorization time varies significantly – hence, the security of a particular key length depends on the target domain.

We discuss the overall impact based on the following aspects:

(1) Accessibility of public keys – how difficult it is for an attacker to obtain the target public key(s) for subsequent factorization attempts;
(2) Total number of factorizable keys found or assumed – as detected by scans of a given usage domain;
(3) Cost to factorize the keys with the lengths actually used in the target domain (as estimated in Table 2);
(4) Implications of a successful factorization – what damage the attacker can cause.

Note that due to the varying parameter M used by the *RSALib* when generating the keys of different lengths, the difficulty of key factorization does not strictly increase with the key length (see Figure 1). Some shorter keys may be actually more difficult to factorize using our method than other longer keys. As an example,

Key size	University cluster (Intel E5-2650 v3@3GHz Q2/2014)	Rented Amazon c4 instance (2x Intel E5-2666 v3@2.90GHz, estimated)	Energy-only price ($0.2/kWh) (Intel E5-2660 v3@2.60GHz, estimated)
512 b	1.93 CPU hours *(verified)*	0.63 hours, $0.063	$0.002
1024 b	97.1 CPU days *(verified)*	31.71 days, $76	$1.78
2048 b	140.8 CPU years	45.98 years, $40,305	$944
3072 b	$2.84 * 10^{25}$ years	$9.28 * 10^{24}$ years, $$8.13 * 10^{27}$	$$1.90 * 10^{26}$
4096 b	$1.28 * 10^{9}$ years	$4.18 * 10^{8}$ years, $$3.66 * 10^{11}$	$$8.58 * 10^{9}$

Table 2: An estimation of factorization times and prices for different key lengths on different types of computational devices. All results are the worst case estimates with expected resources spent being the half of the values shown. The time values marked as *(verified)* were practically verified by factorization of real test keys while others were extrapolated based on a know number of attempts and a time per attempt. The energy consumption was estimated based on the thermal design power (TDP) specifications of Intel Xeon E5-2660 v3 @ 2.60 GHz [42] (note that peak power can be up to 1.5-3x more), time per attempt as benchmarked on Amazon c4 instance, energy price of $0.2/kWh and scaled to 2.90 GHz (as Amazon c4 uses publicly unreleased Intel Xeon E5-2666 v3 clocked at a slightly higher frequency). The university cluster column captures the factorization times as measured by us on a university computational cluster with Intel Xeon E5-2650 v3 @ 3.00 GHz CPUs scaled to a single-core of this CPU. The Amazon c4 instance price corresponds to outsourcing of a single key factorization to Amazon AWS (c4 price is $0.1/hour for a 2-core CPU). We performed benchmark on a c4 instance for a single Coppersmith's computation and extrapolated to number of attempts in the worst case. The energy-only price corresponds to situation when one operates own hardware and wants to factorize so many keys that the price of hardware completely amortizes over all factorized keys. A factorization benchmark on Microsoft Azure was also performed with results roughly comparable to Amazon AWS (+10%).

a 1280-bit key is more difficult to factorize than a 2048-bit key in our setting. It is crucial to survey the precise key lengths as used within the inspected domains. We take advantage of the possibility to quickly detect the key fingerprint, with quick summary of the affected domains in Section 4.1 and in Table 3 and Table 4 followed with additional details for every domain thereafter.

4.1 Summary of results

The electronic identity documents (eIDs) domain is significantly affected. Despite the general difficulty of obtaining relevant datasets with public keys from passports or eIDs that limited our analysis to only four countries, we detected two countries issuing documents with vulnerable keys. The public lookup service of *Estonia* allowed for a random sampling of the public keys of citizens and revealed that more than half of the eIDs of regular citizens are vulnerable and that all keys for e-residents are vulnerable.

The use of two-factor authentication tokens and commit signing is on the rise, yet these approaches are still adopted only by a minority of developers – but usually for the more significant projects. The analysis of the authentication keys of all GitHub developers found several hundreds of vulnerable keys. The developers with vulnerable keys have access to crucial open-source repositories with more than 50,000 stars. Increased scrutiny should be applied to new commits before the affected users replace vulnerable keys.

Trusted Platform Modules (TPMs) provide secure hardware anchor for trusted boot. Although it is difficult to directly extrapolate the overall prevalence of chips with vulnerable keypair generation from our limited sample of 41 laptops with different TPM chips, approximately 24% were producing vulnerable keys, indicating that the domain is significantly affected. As the replacement of a chip alone is very impractical or almost impossible, organizations have to replace the whole laptop, slowing down the recovery from the problem. Importantly, TPM is used not only to facilitate trusted boot, but also to store sensitive secrets like ones necessary to access the Volume Master Key (VMK) for Microsoft BitLocker full disk encryption software [58]. The possibility to factorize TPM's 2048-bit key for "sealed storage" might lead to a recovery of BitLocker's disk decryption key in the configuration using a TPM and a PIN.

The Pretty Good Privacy (PGP) keys used for digital signatures and email encryption are easy to download from PGP keyservers. We detected almost three thousand fingerprinted keys with slightly less than one thousand practically factorizable. The *Yubikey 4* token seems to be the origin for the majority of these keys as hundreds even contain identifying strings in the keyholder information and the date of generation correlates with the release date of this token.

We found only a negligible fraction of vulnerable keys in the TLS/HTTPS domain. However, all 15 unique keys found were tied to different pages with SCADA-related topics, which may point to a single provider of a SCADA remote connection platform.

We did not collect relevant datasets of public keys for authentication tokens implementing PIV or U2F standard but found at least one instance of a widely used token utilizing chips with the affected *RSALib*. Similarly, other devices (e.g., e-health and EMV payment cards) might be impacted by the described vulnerability, although we were not able to verify the impact in such domains.

We encourage the use of our tool for detecting vulnerable keys described in Section 5 and the notification of affected parties if found.

4.2 Electronic identity documents

Various citizen identity documents represent a large area for the application of cryptographic smartcards, such as biometric passports (ePassport, ICAO Doc 9303), eDriver licenses (ISO/IEC 18013) and additional identity documents. Some national IDs are based on the same suite of protocols as ePassports, which are standardized by ICAO 9303 [41]. Other countries have implemented their own suite of protocols, such as the Estonian EstEID [5], the Belgian eID [6] or the Taiwanese ID.

Electronic passports and identity cards utilize digital signatures for: 1) the authentication of stored data (passive authentication); 2) the verification of the genuine origin of the chip inside (active authentication, AA); and 3) the establishment of a secure channel between the passport and the border inspection terminal with mutual authentication (Extended Access Control, EAC-PACE). Additionally, in some instances, the issuing country uses the national IDs for citizen authentication when accessing government services via the Internet.

The suppliers of ePassport implementations typically provide the platform in several possible configurations with different supported algorithms (RSA-based, EC-based) and protocols (EAC-PACE, AA), leaving the choice of the preferred configuration to the issuing country. The use of the *RSALib* is referenced in multiple certification documents of electronic passports of several countries.

We are not aware of any country disclosing publicly the full database of their citizens public keys. A small fraction of countries provide lookup services with significant limitations on the number of queries allowed. We analyzed four different types of digital certificates issued by the country of *Estonia*: a) regular citizenship eID keys (denoted as *esteid*); b) eID keys for electronic use only ("digital certificate of identity", denoted as esteid-*digi*); c) keys for operations from mobile devices (denoted as esteid-*mobiil*); and d) e-resident keys (denoted as esteid-*resident*). For every type, separate authentication (*auth*) and signature (*sign*) 2048-bit RSA keys are available. The keys are used to support various eGovernment services, including VAT forms, private companies management (all types) and voting (*esteid*). In total, we analyzed the keys of approximately 10% of randomly selected citizens. The results showed a mix of on-card and out-of-card key generation. More than half of the analyzed keys were vulnerable for *esteid* and all keys were vulnerable for esteid-*digi* and esteid-*resident*. No vulnerable keys were detected for esteid-*mobiil*. Extrapolation to the whole population results in at least hundreds of thousands of vulnerable keys.

Additionally, we analyzed keys from a limited sample of keys extracted from the physical electronic documents of three other countries and detected one (*Slovakia*) issuing documents with fingerprinted 2048-bit keys.

These results also demonstrate the general difficulty of analyzing the impacted domains – large-scale analysis was possible only for the *Estonian* eID because of the public directory with more than half of the documents found to be vulnerable. The small samples collected for other countries (like *Slovakia*) give only very limited insight – are all other documents vulnerable or only a limited production series given the two vulnerable IDs detected? Or were only documents from non-vulnerable series for other countries inspected?

The possibility of factorizing on-card keys would lead to cloning of legitimate passports or identity cards. The Slovak national ID in question is also deployed in the wider context of an eGovernment system, where the on-chip generated digital signatures serve as a replacement for traditional hand-written signatures.

4.3 Code signing

The digital signing of applications, modules, OS distributions or code is now common. In some cases, application signing is mandatory and enforced by the platform (e.g., Android, iOS, OS drivers) or voluntarily adopted by the developers. GPG signatures can be also used to authenticate commits or tags submitted by developers to a source control system (e.g., GitHub).

4.3.1 GitHub. To access the Git repositories hosted on GitHub, developers can use SSH authentication as an alternative to a password for both read and write permissions. Users may also upload GPG keys for commit signing. The public keys of all users are accessible via the public GitHub API. We analyzed the profiles of almost 25 million GitHub users and found 4.7 M SSH keys in a scan performed in February 2017.

Hundreds of fingerprinted keys were found, including keys with access to very popular repositories with up to 2,000 stars (users bookmarking the project) for user-owned repositories and more than 50,000 stars for organization-owned repositories, including repositories that are very influential in the Internet community. The impact is increased by the fact that some relevant repositories are libraries used in other projects and are essentially trusted by third-party developers.

In total, we found 447 fingerprinted keys. More than half (237) have practically factorizable key length of 2048 bits, with the rest mostly being 4096-bit RSA keys. However, it is not straightforward to determine whether a particular account has write access to repositories not explicitly owned by the account. Similarly, membership in an organization does not guarantee write access to particular repositories. GitHub does not provide this kind of information directly, and the APIs that can be used to derive this information are quite limited. The information can be inferred from an analysis of previously performed commits by the given user. We verified several instances manually and confirmed access with factorizable keys.

We view the overall impact as significant. Luckily, any potential changes made to a repository can be traced back to a particular commit due to the nature of source control systems. Many projects also use commit reviews (e.g., using pull requests), where increased caution should be used until the affected users move to more secure keys.

4.3.2 Maven. The Maven public repository has required developers to sign uploaded artifacts since approximately 2009 [54]. Each developer must be associated with a PGP key that is also publicly reachable from a PGP keyserver. Each artifact is uniquely identified by a tuple (group ID, artifact ID, version). We downloaded the most recent versions of each artifact found in the Maven repository index in April 2017. In total, we downloaded 180,730 artifacts equipped with the Maven index file (pom.xml) – 161,841 had a signature on the pom.xml file. There were 16,959 unique PGP keys found, of which 5 were fingerprinted, all with 4096-bit moduli (not considered practically factorizable by our method). The potentially affected artifacts appear as dependencies only in a few other artifacts. We therefore estimate the impact as small.

4.3.3 Android. We downloaded the 540 most popular Android applications and the 540 top ranking Android games according to the Google Play top charts. The content of the Android application package (APK) is signed with the developer key before being published to the Google Play system. There is no simple way for the developers to change the signing keys; hence, the applications will most likely have used the same keys since the time of the first upload. No fingerprinted keys were detected among the top 540 applications and games in a scan performed in January 2017. The analysis should be also extended to less popular applications. If any vulnerable keys are found in other already established applications, the affected developers may have complications migrating to different signing keys.

4.4 Trusted Platform Modules

Trusted Platform Module (TPM) is a specification created by the Trusted Computing Group [34, 35]. TPMs are cryptographic hardware (usually in form of a chip attached to a motherboard) that provide basic cryptographic functionality. The typical use cases include: a) secure storage of a user's private keys or disk decryption keys; b) maintaining an unspoofable log of applications that were deployed on a target machine via a hash chain (Platform Configuration Registers – PCRs); and c) attestation of the state of the platform to a remote entity by an on-TPM signature of the PCRs. The TPM specification version 1.2 supports only RSA with 2048-bit keys [34].

We analyzed a sample of 41 different laptop models equipped with TPM chips. Six different manufacturers were detected, with chips supplied by *Manufacturer* (acronym IFX) being the most common and found in 10 devices. TPM chips from devices produced before 2013 and with firmware versions[5] between 1.02 and 3.19 do not exhibit a fingerprint and are not factorizable by our method. All chips found in devices introduced in 2013 or later were vulnerable, including both TPM 1.2 and TPM 2.0. In our sample, the fingerprinted keys from the *RSALib* appear earliest in the firmware version 4.32 (however, we had no TPM chip with a version between 3.19 and 4.32 in our sample). All subsequent chip versions, including 5.x and 6.x, were also found to produce vulnerable keys. We hypothesize that the *RSALib* was first used with TPM firmware version 4.x.

There are two important RSA private keys stored inside a TPM – the *Endorsement key* (EK), which is permanently embedded by the chip manufacturer during its production and cannot be changed, and the long-term *Storage Root Key* (SRK), which is generated on-chip when a user claims the TPM ownership. Additionally, dedicated *Attestation Identity Keys* (AIKs) used for Remote Attestation may be generated.

The factorization of the EK compromises the root of trust for chip authentication. An attacker can generate a new keypair outside the TPM and then sign it with the factorized EK; hence, it will be trusted by the remote system (e.g., the company network).

[5]The version of the firmware of the TPM chip does not directly relate to the version of the *RSALib*.

The TPM can hold only a very limited number of private keys directly on the chip. All other private keys are generated inside the TPM but are then wrapped by the SRK and exported outside the TPM. If required, the keys are imported back, unwrapped and used. The factorization of the SRK therefore allows an attacker to decrypt all previously exported wrapped private keys, including the "sealed storage" packages with sensitive information otherwise readable only on the particular machine with the associated AIK keys used for Remote Attestation. If AIK is directly factorized or its value is compromised due to the factorization of the SRK, an attacker is able to forge an attestation report – allowing the attacker to start additional or modified malicious software without being noticed.

The "sealed storage" is also utilized by Microsoft BitLocker full disk encryption software [58] to store a sensitive value required to obtain the Volume Master Key [48, 49]. BitLocker is typically setup together with TPM and an additional secret – either a PIN, a recovery key on a USB token, or both. The possibility to factorize TPM's 2048-bit SRK directly leads to a decryption of an unwrapping key necessary to decrypt the Volume Master Key, thus bypassing the need for TPM to validate the correctness of a PIN value via a dedicated PCR. As a result, an attacker can decrypt a disk from a stolen laptop with a vulnerable TPM if encrypted by BitLocker in TPM+PIN mode (but not in a configuration with an additional USB token). We did not verify the attack in practice due to BitLocker's proprietary storage format and the cost of factorization of a 2048-bit SRK key.

4.5 PGP with cryptographic tokens

The private key as used in Pretty Good Privacy (PGP) [29] is typically a very sensitive long-term secret. If compromised, an attacker can forge new signatures and decrypt all previously captured messages since PGP does not provide forward secrecy. Many users choose to use a cryptographic device that stores and performs private key operations inside a secure environment using an OpenPGP compliant application [62].

A large fraction of public keys used for PGP can be easily downloaded from PGP keyservers [4]. Since the content of PGP key servers is publicly available, the vulnerable keys can be easily identified together with the associated user contact information. We analyzed the state of a PGP keyserver from mid-April 2017 that contained a total of 4.6 M master keys and 4.4 M sub-keys with 1.9 M and 1.7 M, respectively, being RSA keys. We detected 2,892 fingerprinted keys. Of these, two keys are 1024-bit and 954 keys are 2048-bit – both lengths are practically factorizable. Additionally, 86 and 1846 fingerprinted (but not feasibly factorizable by our method) keys of 3072 and 4096-bit lengths, respectively, were detected. Finally, four keys with uncommon lengths of 3008 and 3104-bit were present.

The earliest creation date of a fingerprinted key as obtained from a PGP certificate is 2006, yet only for a single user – we hypothesize this finding was caused by an incorrect system clock. The subsequent observed year is 2009, again with a single user only. 2013 is the earliest year with keys from multiple users.

No key is observed originating in the year 2014, with more fingerprinted keys observed from July 2015 onwards. The date coincides with the official launch of a cryptographic token *Yubikey 4* (further

Domain name	Used length (bits)	Pub. key availability	Misuse
TLS/HTTPS	2048	easy	MitM/eavesdropping
Message security (PGP)	1024/2048	easy	message eavesdropping, forgery
Trusted boot (TPM)	2048	limited	unseal data, forged attestation
Electronic IDs (eID, ePassport)	2048	limited	clone passport, e-gov document forgery
Payment cards (EMV)*	768/960/1024/1182	limited	clone card, fraudulent transaction
Certification authorities (root, intermediate)*	2048 or higher	easy	forged certificates, MitM
Authentication tokens (U2F)	2048 or higher	limited	unauthorized access or operation
Software signing	2048 or bigger	easy	malicious application update
Programmable smartcard (Java Card)	1024-4096	depends on use	depends on use

Table 3: The summary of the impact of key factorization in the different usage domains. The fingerprinted keys were found within all listed domains with exceptions marked with an asterisk (*). No fingerprinted keys were found in the very limited dataset of 13 EMV cards that we collected or for large datasets of browser-trusted root and intermediate CAs.

Domain name	Analyzed datasets	# Vuln. keys/devices	% Vulnerable
Complete/larger-scale datasets			
Certification authorities	all browser-trusted roots (173), level $\leqslant$ 3 intermediates (1,869)	0 keys	0
ePass signing certificates	ICAO Document Signing Certificates, CSCA Master Lists	0 keys	0
Estonian eID	sample of 130,152 randomly selected citizens	71,417 keys	54.87
Estonian mobile eID	sample of 30,471 randomly selected citizens	0 keys	0
Estonian e-residents	sample of 4,414 e-residents	4,414 keys	100
Message security (PGP)	complete PGP key server dump (9 M)	2,892 keys	0.03
Software signing (GitHub)	SSH keys for GitHub developers (4.7 M)	447 keys	0.01
Software signing (Maven)	signing keys for all public Maven artifacts	5 keys	0.003
TLS/HTTPS	complete IPv4 scan, Certificate Transparency	15 keys	<0.001
Trusted boot (TPM)	41 laptops with different chips by 6 TPM manufacturers	10 devices	24.39
Limited, custom-collected datasets			
Payment cards (EMV)	13 cards from 4 EU countries, 6 with *Manufacturer* chip	0 keys	0
Programmable smartcard	25 cards from JCAlgTest.org database, 6 with *Manufacturer* chip	2 cards	8.67
Software signing (Android)	1,080 top ranking applications and games	0 keys	0

Table 4: The summary of the number and fraction of vulnerable keys detected in different domains. The domains are ordered lexicographically and separated into two groups based on the representativeness of inspected datasets.

denoted as *Token*). This hints that *Token* is the major source of the fingerprinted keys in the PGP dataset Out of 2,892 fingerprinted keys, 436 even contain some form of *Token*-related identification in the *User ID* string (154 being master keys with the rest being sub-keys). Of these, no key with a length shorter than 2048-bit is present, 96 keys are 2048-bit and 340 keys have a length of 4096 bits. Given that an older version of *Token* is not producing fingerprinted keys, all these keys were likely generated by the newer version of *Token*.

The *Token* vendor recommends generating a keypair outside the token (for example, using OpenSSL) and importing it to facilitate private key recovery after a potential token failure. Interestingly, such advice seems not to have been followed by a significant number of users (the users who followed this advice are not detected by our fingerprinting method as their keys have no fingerprint).

The evidence for other devices (not produced directly by the *Manufacturer*) generating fingerprinted keys also shows that the *RSALib* is provided to external parties developing for the *Manufacturer* hardware.

We would like to stress that not all key lengths generated with *Token* are immediately practically factorizable by our method. *Token* can generate and use RSA keys up to 4096 bits long, which may be one of the appeals of the device – given the lack of other available smartcards supporting key lengths exceeding 2048 bits. Indeed, the analysis of the fingerprinted PGP keys with respect to the used length shows a strong user preference for 4096-bit keys. *Token* can also generate less common key lengths including 3936-bit RSA where our attack is not directly applicable, as seen in Figure 1. The majority of the *Token* users on this domain therefore should not be imminently affected by direct factorization using our attack, but we urge the generation of fresh keys – in light of potential further improvement of an attack.

4.6 TLS and SCADA-related keys

We used our fingerprinting method on two large datasets of public key certificates, used (mostly) to secure Internet TLS connections. One dataset originates from a periodic scan of the whole IPv4 address space between 2012 and 2017 [28] collected from servers listening on port 443 and configured to prefer RSA signatures. The second dataset comes from the Certificate Transparency logs maintained by Google [33] (CT logs maintained by Google included in Google Chrome, date 2017-04-25). In total, we analyzed more than 100 million certificates.

Despite the relatively large number of keys, we only found 15 distinct fingerprinted keys – four were 1024 bits long and eleven with 2048 bits – used in tens of different certificates. Surprisingly, almost all these certificates contain the string "SCADA" in the common name field (probably referring to Supervisory Control and Data Acquisition systems) or a URL leading to a website related to an industrial monitoring system, or both. As a result, we hypothesize that there is at least one provider of a remote connection platform with a focus on SCADA systems. It is not clear to us whether the interfaces are linked to real industrial systems since administrators of such systems may want to limit the access from the Internet. Hence, there might exist more systems with administration interfaces protected by vulnerable keys, but deployed on local networks.

Interestingly, all 15 keys contain the inspected fingerprint, but the majority of values of the most significant byte (MSB) of their moduli are significantly outside the range observed in the *RSALib* (the MSB of keys produced by the inspected smartcards and TPMs always falls in the interval `0x90-0xA8`). This finding suggests the existence of a different implementation of the prime construction algorithm with the same structure but a different modification of the most significant bits.

The factorization of the key of a TLS server trivially leads to numerous powerful attacks: server impersonation, active man-in-the-middle attack or passive decryption of the content of the communication when the connection establishment lacks forward secrecy. Overall, the impact on the public portion of the Internet seems to be only very marginal due to the small number of detected vulnerable TLS keys. However, the potentially significant impact for the entry points of some SCADA services should not be neglected.

4.7 Certification authorities

The presence of vulnerable keys belonging to certification authorities would magnify the impact due to the possibility of key certificate forgery. We therefore examined two significant usage domains.

Browser-trusted certificates. We examined the certificates of root certification authorities stored in Mozilla Firefox as browser-trusted roots (158 certificates) and in Ubuntu 16.04 (173 certificates). The intermediate authorities of level 1 (1,016 total), level 2 (832 total) and level 3 (21 total) as extracted from TLS scans were also analyzed. No fingerprinted keys were detected as of May 2017.

ICAO signing certificates. We analyzed the collection of Document Signing Certificates (DSCs) of the ICAO ePassport database (version 2044) containing 8,496 certificates, and the collection of CSCA Master Lists (version 84) with 616 certificates. We also inspected the publicly available national certificates (e.g., Belgium, Estonia, Germany, Switzerland) [43, 68] available as of May 2017. Fortunately, no vulnerable keys were found in either dataset, as the occurrence of such a certificate would lead to the possibility of impersonating an inspection terminal or forging electronic document data.

4.8 Generic Java Card platform

Smartcards using the Java Card platform [59] have two principal configurations: 1) an open, fully programmable platform where the users develop and upload their own applications; and 2) Java Card-based systems closed from the point of view of cryptography (e.g., banking EMV or SIM cards). Here, we focus on the former configuration.

The prevalence of the *RSALib* in the area of programmable smartcards is notoriously difficult to estimate. Not all smartcards based on the *Manufacturer*'s hardware are vulnerable, as the vulnerability stems from the deployed cryptographic library and not from the hardware design itself. Many vendors use the bare hardware (e.g., SLE78 chip) and choose not to deploy the *RSALib* in question. In such a case, the implementation of the higher-level cryptographic functions (including RSA keypair generation) is done by the company that builds on the hardware produced by the *Manufacturer*. Although the vulnerable keys have a strong fingerprint that can be easily verified, the real problem (for impact assessment) lies in obtaining sample public keys. No representative public databases (comparable to those for TLS and PGP) are available.

Our analysis is based on smartcards from 10 different platform providers (Axalto, Feitian, G&D, Gemalto, Infineon, JavaCardOS, NXP, Oberthur, Softlock and Yubico) as recorded by the *JCAlgTest* database [60]. The chip manufacturer (*ICFabricator* property) and the manufacturing date (*ICFabricationDate*) can be obtained from the Card Production Life Cycle (CPLC) information as defined by the GlobalPlatform specification [32].

Out of the 63 different cards included, 25 cards are listed with the provided CPLC information: 16x NXP (*ICFabricator* = 4790), 6x Infineon (4090), 1x Samsung (4250) and 2x unknown (2050 and 4070). Out of six cards with a *Manufacturer* chip, two produce fingerprinted keys. The *ICFabricationDate* property indicates the years of manufacture to be 2012 and 2015. Hence, our estimate of the prevalence of the vulnerability is confirmed again since it corresponds to the situation observed in TPM chips.

The full impact of the vulnerability will depend entirely on the scenario in which the cards are actually used. The large number of already fabricated and distributed smartcards may hinder the potential for a recall of the product from the market. The card operating system and the base libraries are stored in read-only memory and cannot be updated by the user to remove the vulnerability once a card is deployed. We expect to see the cards for a rather long time (several years) before all the vulnerable cards are eventually sold out, especially when dealing with low volume markets. The buyers should check the cards for the presence of fingerprinted keys and/or opt for longer key lengths if they are supported by the card hardware.

4.9 Other domains

The smartcards are also used in many other domains than those surveyed here in the previous sections, including authentication tokens (e.g., U2F [70]); e-health cards to authenticate both patients and medical staff to access medical records or personal identity verification cards (FIPS 201 PIV [61]); and electronic payment cards (EMV).

The chip-based payment cards used world-wide are backed by a set of protocols specified under the EMV standard [52], which is currently maintained by the EMV consortium. The *RSALib* was approved for use in EMV cards by EMVCo [2, 3], and we found several references to it in related certification reports. However, we are not aware of any public dataset of keys originating from EMV cards. We collected a tiny sample of RSA keys from 13 payment cards issued by different banks in four European countries. Although 6 cards reported chips produced by the *Manufacturer*, none of them contained the distinctive fingerprint, meaning that the RSA key generation method implemented by the *RSALib* was not used in either one.

If used, the potential impact of factorizable keys would be particularly damaging to EMV cards due to the generally short RSA key lengths used. Short keys are often used for legacy reasons or to improve the usability of payments by the shorter time required to authorize the transaction (especially relevant for contactless payments). Out of the 13 cards inspected, we observed the following bit lengths of ICC keys: 768 (3x), 896 (4x), 960 (1x) and 1024 (5x).

We recommend analyzing the keys used in a particular scenario with the provided fingerprint detection tool and following the recommendations given in Section 5.

5 MITIGATION AND DISCLOSURE

We propose a mitigation of the attack impacts and report on the process of responsible disclosure to the *Manufacturer*.

5.1 Mitigation

Mitigation can be performed on multiple levels. Inarguably, algorithm replacement is the best long-term mitigation method. However, changing the algorithm requires updating firmware – which is usually not possible in already deployed devices like smartcards or TPMs with code stored in read-only memory. Other options are available even within the hardware device with the vulnerable version of the *RSALib* with some caveats. New keys can be still generated on the device if they are configured to use key lengths not directly affected by our method (yet still with a reduced security margin), or keypairs could be generated by another library (outside the device) and then imported to the device. If the potentially vulnerable keys remain deployed, their usage scenario can be supplemented with additional risk management.

5.1.1 Changes to the algorithm. The library could adopt an approach common in open-source libraries – instead of constructing candidates for the primes, they are generated randomly and their value is incremented until a prime is found. Other alternative constructions exist, such as provable or safe primes, as described in the NIST FIPS 186-4 standard [46]. We noticed a certain similarity between the algorithm of the *Manufacturer* and an algorithm published by Joye and Paillier [44] focused on key generation on smartcards. The key difference seems to be the fact that the *RSALib* uses a constant value in the generator (65537), while in the paper, the value is always chosen randomly using a unit generation algorithm [44, Figure 4]. The approach in the paper [44] is *not* affected by the same vulnerability.

Note that due to the nature of deployment of the *RSALib*, the devices already in use cannot be updated. The *RSALib* is often stored in a read-only on-chip memory with no possibility to distribute and apply a fix after deployment.

5.1.2 Importing a secure keypair. A secure RSA keypair can be generated in another cryptographic library and then imported to the affected device. We are not aware of any vulnerability in *Manufacturer* devices as far as the use of securely generated keys is concerned. Coincidentally, the import of externally generated keypair is even recommended by Yubico vendor [77], although for the purpose of private key backup.

5.1.3 Use of less affected key lengths. As discussed in Section 2, we consider 512, 1024 and 2048-bit keys to be insecure. Due to design choices made by the manufacturer, it appears that 3072-bit keys are seemingly less affected by our method than 4096-bit RSA though with a significantly reduced security margin. Our attack is inefficient or directly inapplicable when applied to some quite uncommon key lengths (such as 1952 bits or 3936 bits). Hence, we recommend limiting the choice of the key lengths to the seemingly unaffected keys if the usage of the vulnerable chips with on-chip generated keys is absolutely unavoidable. Note however, that these keys still suffer from significant entropy loss. If a somewhat "standard" key length is required, we recommend switching to 3072-bit keys.

We also suggest caution when using the fingerprinted 4096-bit keys, even though our method is not practical for their factorization at the moment (requiring $1.28 * 10^9$ CPU-years). The strongest possible key length with respect to the general factorization methods and our attack is 3936-bit RSA. If a device supports at most 2048-bit keys, the key length of 1952 bits is the most secure option (see Figure 1).

5.1.4 Additional risk management. The use of potentially vulnerable keys (especially 2048-bit keys requiring feasible yet still significant computational power) can be amended with additional scrutiny to perform supplementary risk management. The presence of the fingerprint is an advantage in this scenario since the public keys can be quickly tested to decide when to apply additional measures by the cooperating system.

5.2 Future prevention and analysis

The impacts of the documented vulnerability may serve as cases supporting the need for future systematic changes and deeper additional analyses, not limited just to the library in question.

5.2.1 Preventing the single point of failure. The described problem would be mitigated if not a single but two or even more independent implementations were used to generate the RSA keypair. More generally, a secure multi-party protocol can be utilized to remove the single point of failure, not only during the keypair generation, but also during its use. The general goal is to provide tolerance against up to k out of t misbehaving (either faulty or intentionally malicious) participants [26]. Multiple protocols based on common cryptographic primitives like RSA, Diffie-Hellman or Elliptic curve cryptography were proposed in literature [15, 31, 37, 72]. Such approaches protect not only against an intentionally malicious party, but also against unintentional mistakes weakening the resulting key. The area of collaborative RSA keypair generation is well studied with the primary goal to generate parts (shares) of RSA keypairs, yet not to reveal the factorization of the resulting modulus N, until all or a specified number (threshold) of parties cooperate.

Gilboa's threshold RSA signature scheme [31] requires collaboration during every signature operation, introducing protocol changes. A more efficient generation method by Straub based on 3-prime RSA [72] is not suitable for use by smartcards that implement offline signature generation with limited APIs, typically exposing only standard 2-prime RSA operations. Moreover, protocols securing against active adversaries, like that described in Hazay et al. [37], are time-consuming even on standard CPUs while having prohibitively long keypair generation phases on performance-limited hardware. Parsovs proposed a collaborative method that splits key generation between card manufacturer and cardholder [63]. The resulting 4-prime 4096-bit RSA key is generated from two 2048-bit parts during an interactive protocol executed before the card's first use, limiting the necessity to trust a vendor with the generation of the whole keypair, as well as removing the single point of failure.

Gennaro et al. proposed a distributed key generation algorithm for discrete-log cryptosystems (not directly applicable to RSA) [30], with extensions to provably secure distributed Schnorr signatures [71] and with the implementation shown to be efficient enough to run on cryptographic smartcards [56] as a mitigation of hardware Trojans.

Note that all the methods described above require changes to user interfaces and protocols and are therefore less suitable for legacy systems. However, a systematic adoption of secure multiparty protocols, instead of relying on a single vendor and implementation, can provide a significant overall increase of security of a system.

5.2.2 Analysis of other limited devices. The need for fast keypair generation on limited hardware naturally leads to a search for alternative methods for finding completely random primes. The generation method of Joye and Paillier [44] is one example. Therefore, other modifications (with respect to [44]) or completely different methods may have been adopted by other hardware vendors. We did not detect any deviances in cards from 5 other manufacturers using our fingerprinting method. However, even a minor change to unit generation used in *RSALib* will suppress the bias that is detectable by our method (e.g., generators for p and q other than 65537), yet these changes will not automatically result in keys being secure against variations of our attack. The search for alternative detection techniques as well as attack variations represents possible future work.

5.3 Responsible disclosure

Disclosure of this vulnerability was made to *Manufacturer* in the beginning of February 2017 together with the tools demonstrating fingerprinting capabilities and practical factorization. The vulnerability was subsequently confirmed with further notification of the affected parties by *Manufacturer*.

We made public disclosure of the discovered issue in the middle of October 2017 together with the release a tool for fingerprint detection for provided public keys to facilitate a quick assessment of the presence of the vulnerability for end-users. The full details of the attack are published in this paper.

For the time being, we are not releasing our source code of the factorization algorithm. We believe that honest parties can make their own implementation based on our description.

6 RELATED WORK

The generation of RSA keys and attacks on them are the two main areas related to this work. Besides attacks on the messages (e.g., padding oracle [11, 16, 19] or related messages [25, 76]), most attacks aim to deduce the private key from the corresponding public key. The attacks can be divided into two classes based on the assumptions about the key: 1) No additional information – methods such as Pollard p-1 [65], Pollard Rho [18, 66], and a class of several sieving methods (e.g., NFS, GNFS); 2) Partial information – low private or public exponent [13, 14, 24, 74], implementation and side-channel attacks, and attacks based on Coppersmith's method [23].

The usage of generic attacks is limited to small RSA keys due to their exponential time complexity (the current record for a general 768-bit RSA [47] was broken using NFS). Only attacks from the second class are known to be used to break RSA moduli used in practice. Side-channel attacks (e.g., timing attacks, power analysis) are out of the scope of this work since they require active access to the device performing the RSA computation. Except for Wiener's attack [74] for a small private exponent, other notable attacks belong to the same class as Coppersmith's attack.

In 2012, two independent teams [38, 50] analyzed RSA public keys on the Internet. The teams analyzed several millions of widespread keys in network devices such as keys in SSL certificates, SSH host keys and PGP keys. These teams observed that a small portion (0.5% of TLS, 1% of SSH) of public RSA keys shared prime factors. Due to insufficient entropy (e.g., SSL keys were generated by low-powered devices with no source of entropy) during the generation process, these keys can be trivially factorized using GCD. In 2013, Bernstein et al. [9] analyzed the "Citizen Digital Certificate" database of 3.2 million public RSA keys generated by smartcards used as the national IDs of Taiwanese citizens. In addition to recovering 184 keys that shared primes using a batch GCD computation, the authors adapted Coppersmith's algorithm and computed an additional 81 private keys. To our knowledge, this is the only practical application of Coppersmith's method to attack real RSA keys prior to our attack. Coppersmith's algorithm can be viewed as a universal tool for attacking RSA keys generated with improperly chosen parameters or originating from a faulty implementation. The algorithm was adapted for various scenarios where some bits of a factor, of the private exponent or of the message are known [12]. The factorization of moduli with known high [22] or low [24] bits of a factor were among the first variants of the method. A nice overview of these methods can be found in [57].

The generation of RSA keys is described in several standards (e.g., FIPS 186-4 [46], IEEE 1363-2000 [1] – see [53] for an overview), many having different requirements for the form of the primes. One feature is common to all these standards – the primes should be generated randomly using a large amount of entropy. In addition to specialized construction methods (e.g., provable primes), the generation of RSA primes is typically performed in several iterations, repeating two fundamental steps: a random candidate is generated and then tested for primality. Since the primality test is a time-consuming process, several authors have proposed various speedups for the candidate generation process ([17, 45, 55], see [44] for an overview of such methods). The current state of the art focused on constrained devices is described in [44], where the authors decreased the number of primality tests with a negligible loss of entropy (0.5 bits).

7 CONCLUSIONS

We presented a cautionary case of a vulnerable prime selection algorithm adopted in RSA key generation in a widely used security library of a cryptographic hardware manufacturer found in NIST FIPS 140-2 and CC EAL 5+ certified devices. Optimizations that were motivated by a higher performance in the key generation process have inadvertently led to significantly weakened security of the produced keys. The primes are constructed with a specific structure that makes the factorization of the resulting RSA keys of many lengths (including 1024 and 2048 bits) practically feasible with only the knowledge of the public modulus. Worse still, the keys carry a strong fingerprint, making them easily identifiable in the wild. The factorization method is based on our extension of the Howgrave-Graham refinement of Coppersmith's method.

To quantify and mitigate the impacts of this vulnerability, we investigated multiple domains where the RSA algorithm is deployed. Based on the specific structure of the primes, we devised a very fast algorithm to identify all vulnerable keys even in very large datasets, such as TLS or Certificate Transparency. Where public datasets were missing (eID, TPM, etc.), we attempted to collect some keys on our own. The results confirmed the use of the *RSALib* that produces vulnerable RSA keys across many domains.

There is mounting evidence that prime generation is a critical part of implementations that designers and developers struggle with. Authoritative design notes for robust approaches should be produced and disseminated. Developers must follow existing standards without modifications.

Our work highlights the dangers of keeping the design secret and the implementation closed-source, even if both are thoroughly analyzed and certified by experts. The lack of public information causes a delay in the discovery of flaws (and hinders the process of checking for them), thereby increasing the number of already deployed and affected devices at the time of detection.

The certification process counter-intuitively "rewards" the secrecy of design by additional certification "points" when an implementation is difficult for potential attackers to obtain - thus favoring security by obscurity. Relevant certification bodies might want to reconsider such an approach in favor of open implementations and specifications. Secrecy may increase the difficulty of spotting a flaw (above the capability of some attackers) but may also increase the impacts of the flaw due to the later discovery thereof.

ACKNOWLEDGMENTS

We would like to thank all anonymous reviewers for very enlightening and useful comments as well as our colleagues for fruitful discussions, especially Vladimir Sedlacek and Martin Ukrop. We acknowledge the support of the Czech Science Foundation under project GA16-08565S. The access to the computing and storage resources of National Grid Infrastructure MetaCentrum (LM2010005) is also greatly appreciated. Vashek Matyas thanks Red Hat Czech and CyLab, Carnegie Mellon University for a supportive sabbatical environment during some of his work on this paper.

REFERENCES

[1] 2000. IEEE Std 1363-2000: IEEE Standard – Specifications for Public-Key Cryptography. IEEE. https://books.google.cz/books?id=KKc8nQAACAAJ cit. [2017-08-28].

[2] 2017. EMVCo Product Approval, ICCN0163, Master component: M7892 A22/B11, 11 Jan 2012. (2017). https://www.emvco.com/loa_se/EMVCo_ICCN0163_R_02_2017.pdf cit. [2017-05-11].

[3] 2017. EMVCo Product Approval, ICCN0200, Master component: M7893 B11, 20 Dec 2013. (2017). https://www.emvco.com/loa_se/EMVCo_ICCN0200_R_02_2017.pdf cit. [2017-05-11].

[4] 2017. PGP keydump from April 19, 2017. (2017). http://pgp.key-server.io/dump/ cit. [2017-04-19].

[5] Trüb Baltic AS. 2017. Estonian Electronic ID card application specification, EstEID v. 3.5. http://www.id.ee/public/TB-SPEC-EstEID-Chip-App-v3.5-20170314.pdf cit. [2017-08-28].

[6] Axalto and Zetes. 2004. Public user specification BelPic application v2.0. http://www.foo.be/eID/opensc-belgium/BEID-CardSpecs-v2.0.0.pdf cit. [2017-08-28].

[7] Luciano Bello. 2008. DSA-1571-1 openssl – predictable random number generator. (2008). http://www.debian.org/security/2008/dsa-1571 cit. [2017-08-28].

[8] Elwyn R. Berlekamp. 1967. Factoring Polynomials Over Finite Fields. *Bell System Technical Journal* 46, 8 (1967), 1853–1859.

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

[10] Daniel J. Bernstein, Nadia Heninger, and Tanja Lange. 2012. Batch gcd. (2012). http://facthacks.cr.yp.to/batchgcd.html cit. [2017-08-28].

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

[12] Daniel Bleichenbacher and Alexander May. 2006. New Attacks on RSA with Small Secret CRT-Exponents. In *Public Key Cryptography - PKC 2006*. Springer-Verlag, 1–13.

[13] Johannes Blömer and Alexander May. 2003. New Partial Key Exposure Attacks on RSA. In *Advances in Cryptology – CRYPTO 2003*. Springer-Verlag, 27–43.

[14] Dan Boneh and Glenn Durfee. 1999. Cryptanalysis of RSA with Private Key d Less than $N^{0.292}$. In *Advances in Cryptology — EUROCRYPT '99*. Springer-Verlag, 1–11.

[15] Dan Boneh and Matthew Franklin. 1997. Efficient generation of shared RSA keys. In *Advances in Cryptology-CRYPTO'97*. Springer, 425.

[16] Matteo Bortolozzo, Giovanni Marchetto, Riccardo Focardi, and Graham Steel. 2009. Secure your PKCS#11 token against API attacks!. In *3rd International Workshop on Analysis of Security APIs (ASA-3)*.

[17] Jorgen Brandt, Ivan Damgård, and Peter Landrock. 1993. Speeding up prime number generation. In *Advances in Cryptology — ASIACRYPT '91*. Springer-Verlag, 440–449.

[18] Richard P. Brent. 1980. An improved Monte Carlo factorization algorithm. *BIT Numerical Mathematics* 20, 2 (1980), 176–184.

[19] Daniel R. L. Brown. 2005. A Weak-Randomizer Attack on RSA-OAEP with e = 3. (2005). http://eprint.iacr.org/2005/189 cit. [2017-08-28].

[20] BSI. 2015. Certification Report, BSI-DSZ-CC-0782-V2-2015, Infineon Security Controller M7892 B11 with optional RSA2048/4096 v1.02.013, EC v1.02.013,SHA-2 v1.01 and Toolbox v1.02.013 libraries and with specific IC dedicated software (firmware), v1.0,. https://www.commoncriteriaportal.org/files/epfiles/0782V2a_pdf.pdf cit. [2017-08-28].

[21] David G. Cantor and Hans Zassenhaus. 1981. A New Algorithm for Factoring Polynomials Over Finite Fields. *Math. Comp.* 36, 154 (1981), 587–592.

[22] Don Coppersmith. 1996. Finding a Small Root of a Bivariate Integer Equation; Factoring with High Bits Known. In *Proceedings of EUROCRYPT'96*. Springer-Verlag, 178–189.

[23] Don Coppersmith. 1996. Finding a Small Root of a Univariate Modular Equation. In *Advances in Cryptology — EUROCRYPT '96*. Springer-Verlag, 155–165.

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

[25] Don Coppersmith, Matthew Franklin, Jacques Patarin, and Michael Reiter. 1996. Low-Exponent RSA with Related Messages. Springer-Verlag, 1–9.

[26] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. 2005. Multiparty computation, an introduction. *Contemporary cryptology* (2005), 41–87.

[27] The FPLLL development team. 2016. fplll, a lattice reduction library. (2016). https://github.com/fplll/fplll cit. [2017-08-28].

[28] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and Alex Halderman. 2015. A search engine backed by Internet-wide scanning. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 542–553.

[29] Simson Garfinkel. 1995. *PGP: pretty good privacy*. O'Reilly Media, Inc. ISBN 978-1-56592-098-9.

[30] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. 1999. Secure distributed key generation for discrete-log based cryptosystems. In *Eurocrypt*, Vol. 99. Springer, 295–310.

[31] Niv Gilboa. 1999. Two party RSA key generation. In *Annual International Cryptology Conference*. Springer, 116–129.

[32] GlobalPlatform. 2006. Card Specification Version 2.2. https://www.globalplatform.org/specificationscard.asp cit. [2017-08-28].

[33] Google. 2017. Certificate Transparency logs from April 25, 2017. (2017). https://www.certificate-transparency.org cit. [2017-05-25].

[34] Trusted Computing Group. 2006. TPM Main Specification Version 1.2, Revision 94. https://trustedcomputinggroup.org/tpm-main-specification/ cit. [2017-08-28].

[35] Trusted Computing Group. 2011. TPM Main Specification Level 2 Version 1.2, Revision 116. https://trustedcomputinggroup.org/tpm-main-specification/ cit. [2017-08-28].

[36] Marcella Hastings, Joshua Fried, and Nadia Heninger. 2016. Weak Keys Remain Widespread in Network Devices. In *Proceedings of the 2016 ACM on Internet Measurement Conference*. ACM, 49–63.

[37] Carmit Hazay, Gert Læssøe Mikkelsen, Tal Rabin, and Tomas Toft. 2012. Efficient RSA Key Generation and Threshold Paillier in the Two-Party Setting. In *CT-RSA*. Springer, 313–331.

[38] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. 2012. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *The 21st USENIX Security Symposium (USENIX Security 12)*. USENIX, 205–220.

[39] Jens Hermans, Michael Schneider, Johannes Buchmann, Frederik Vercauteren, and Bart Preneel. 2010. Parallel Shortest Lattice Vector Enumeration on Graphics Cards. In *Progress in Cryptology – AFRICACRYPT 2010*. Springer-Verlag, 52–68.

[40] Nick Howgrave-Graham. 1997. Finding Small Roots of Univariate Modular Equations Revisited. In *Proceedings of the 6th IMA International Conference on Cryptography and Coding*. Springer-Verlag, 131–142.

[41] ICAO. 2006. ICAO Doc 9303, Machine Readable Travel Documents.

[42] Intel. 2014. Intel Xeon Processor E5-2660 v3 CPU specification. Intel. https://ark.intel.com/products/81706/Intel-Xeon-Processor-E5-2660-v3-25M-Cache-2_60-GHz cit. [2017-08-28].

[43] JMRTD. 2017. Certificates for document validation http://jmrtd.org/certificates.shtml. cit. [2017-08-28].

[44] Marc Joye and Pascal Paillier. 2006. Fast Generation of Prime Numbers on Portable Devices: An Update. In *Cryptographic Hardware and Embedded Systems - CHES 2006*. Springer-Verlag, 160–173.

[45] Marc Joye, Pascal Paillier, and Serge Vaudenay. 2000. Efficient Generation of Prime Numbers. In *Cryptographic Hardware and Embedded Systems — CHES 2000*. Springer-Verlag, 340–354.

[46] Cameron F. Kerry and Romine Charles. 2013. FIPS PUB 186. Federal Information Processing Standards Publication, Digital Signature Standard (DSS). (2013).

[47] 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. 2010. Factorization of a 768-bit RSA Modulus. In *Proceedings of the 30th Annual Conference on Advances in Cryptology (CRYPTO'10)*. Springer-Verlag, 333–350.

[48] Jesse D. Kornblum. 2009. Implementing BitLocker Drive Encryption for forensic analysis. *Digital Investigation* 5, 3 (2009), 75 – 84. DOI: http://dx.doi.org/10.1016/j.diin.2009.01.001

[49] Nitin Kumar and Vipin Kumar. 2008. Analysis of Window Vista Bitlocker Drive Encryption. NVLabs. http://www.nvlabs.in/uploads/projects/nvbit/nvbit_bitlocker_presentation.pdf cit. [2017-08-28].

[50] Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung, and Christophe Wachter. 2012. Public Keys. In *Advances in Cryptology - Crypto 2012 (Lecture Notes in Computer Science)*, Vol. 7417. Springer-Verlag, 626–642.

[51] Arjen Klaas Lenstra, Hendrik Willem Lenstra, and László Lovász. 1982. Factoring polynomials with rational coefficients. *Math. Ann.* 261, 4 (1982), 515–534.

[52] EMVCo LLC. 2011. EMV Integrated Circuit Card Specifications for Payment Systems. http://www.emvco.com/specifications.aspx?id=223 cit. [2017-08-28].

[53] Daniel Loebenberger and Michael Nüsken. 2011. Analyzing standards for RSA integers. *CoRR* abs/1104.4356 (2011). http://arxiv.org/abs/1104.4356

[54] Sander Mak. 2012. Verify dependencies using PGP. http://branchandbound.net/blog/security/2012/08/verify-dependencies-using-pgp/ cit. [2017-08-28].

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

[56] Vasilios Mavroudis, Andrea Cerulli, Petr Svenda, Dan Cvrcek, Dusan Klinec, and George Danezis. 2017. A Touch of Evil: High-Assurance Cryptographic Hardware from Untrusted Components. In *to appear at 24th ACM Conference on Computer and Communications Security (CCS'2017)*. ACM.

[57] Alexander May. 2010. Using LLL-Reduction for Solving RSA and Factorization Problems. In *The LLL Algorithm: Survey and Applications*. Springer-Verlag, 315–348.

[58] Microsoft. 2013. Technet: BitLocker Overview. Microsoft. https://technet.microsoft.com/en-us/library/hh831713(v=ws.11).aspx

[59] Sun Microsystems. 2006. Application Programming Interface Java Card Platform, Version 2.2.2.

[60] CRoCS MU. 2017. JCAlgTest: JavaCard Algorithm Test. (2017). https://jcalgtest.org/ cit. [2017-08-28].

[61] NIST. 2013. FIPS PUB 201-2: Personal Identity Verification (PIV) of Federal Employees and Contractors. NIST. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.201-2.pdf

[62] OpenPGP 2007. RFC4880: OpenPGP Message Format. (2007). https://tools.ietf.org/html/rfc4880 cit. [2017-08-28].

[63] Arnis Parsovs. 2014. Identity Card Key Generation in the Malicious Card Issuer Model. In *MTAT Research seminar report*. https://courses.cs.ut.ee/MTAT.07.022/2014_spring/uploads/Main/arnis-report-s14.pdf cit. [2017-08-28].

[64] Stephen Pohlig and Martin Hellman. 2006. An Improved Algorithm for Computing Logarithms over GF(p) and Its Cryptographic Significance. *IEEE Transactions on Information Theory* 24, 1 (2006), 106–110.

[65] John M. Pollard. 1974. Theorems on factorization and primality testing. *Mathematical Proceedings of the Cambridge Philosophical Society* 76, 3 (1974), 521–528.

[66] John M. Pollard. 1975. A Monte Carlo method for factorization. *BIT Numerical Mathematics* 15, 3 (1975), 331–334.

[67] John M. Pollard. 1993. *Factoring with cubic integers*. Springer-Verlag, 4–10.

[68] rfidiot.org. 2017. e-passport Certificates. http://rfidiot.org/certificates.html cit. [2017-08-28].

[69] Ronald L Rivest, Adi Shamir, and Leonard Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* 21, 2 (1978), 120–126.

[70] Sampath Srinivas, Dirk Balfanz, Eric Tiffany, and Alexei Czeskis. 2017. Universal 2nd Factor (U2F) Overview. In *FIDO Alliance Proposed Standard v1.2 (11 April 2017)*. FIDO Alliance.

[71] Douglas R Stinson and Reto Strobl. 2001. Provably secure distributed Schnorr signatures and a (t, n) threshold scheme for implicit certificates. In *ACISP*, Vol. 1. Springer, 417–434.

[72] Tobias Straub. 2003. Efficient two party multi-prime RSA key generation. In *Proceedings of IASTED International Conference on Communication, Network, and Information Security*. ACTA Press, 100–105.

[73] Petr Švenda, Matúš Nemec, Peter Sekan, Rudolf Kvašňovský, David Formánek, David Komárek, and Vashek Matyáš. 2016. The Million-Key Question – Investigating the Origins of RSA Public Keys. In *The 25th USENIX Security Symposium (USENIX Security '16)*. USENIX, 893–910.

[74] Michael J. Wiener. 1990. Cryptanalysis of short RSA secret exponents. *IEEE Transactions on Information Theory* 36 (1990), 553–558.

[75] David Wong. 2015. Implementation of Coppersmith attack (RSA attack using lattice reductions). (2015). https://www.cryptologie.net/article/222/implementation-of-coppersmith-attack-rsa-attack-using-lattice-reductions/ cit. [2017-08-28].

[76] Oded Yacobi and Yacov Yacobi. 2005. A New Related Message Attack on RSA. In *Public Key Cryptography – PKC 2005 (Lecture Notes in Computer Science)*, Vol. 3386. Springer-Verlag, 1–8.

[77] Yubico. 2017. PGP, Importing keys. https://developers.yubico.com/PGP/Importing_keys.html cit. [2017-08-28].