UNIVERSITY OF TARTU
Institute of Computer Science
Cybersecurity Curriculum

Bruno Produit

# Optimization of the ROCA (CVE-2017-15361) Attack

Master's Thesis (30 ECTS)

Supervisor: Arnis Paršovs, MSc

Tartu 2019

## Optimization of the ROCA (CVE-2017-15361) Attack

**Abstract:**
In 2017, Czech researchers found the vulnerability CVE-2017-15361 (the ROCA attack) in Infineon's proprietary RSA key generation algorithm. The researchers found that 2048-bit RSA key can be factored in only 140.8 CPU-years in the worst case scenario. The algorithm turned out to be used by 750 000 Estonian ID-cards. In this thesis, we implemented the ROCA attack and, based on the properties observed from the keys generated by the affected smartcards, found further optimizations which allow to improve the original attack from 140.8 CPU-years to 35.2 CPU-years for 90% of the keys and 70.4 CPU-years for the remaining 10% of the keys. As additional contribution, we provide a parallelized version of the attack that can be executed on an HPC.

**Keywords:** Cryptography, smartcard, ROCA, RSA, Coppersmith, Factoring, Entropy, Cyber-security, High-performance computing

**CERCS:** P170, Computer science, numerical analysis, systems, control

## ROCA rünnaku optimiseerimine (CVE-2017-15361)

**Lühikokkuvõte:**
2017. aastal avastasid Tšehhi teadlased Infineoni loodud RSA võtmete genereerimis algoritmist haavatavuse CVE-2017-15361 (ROCA rünnak). Leiti, et Infineoni algoritmiga genereeritud 2048-bitiseid võtmeid on võimalik faktoriseerida halvimal juhul kõigest 140.8 CPU aastaga. Antud algortimi kasutades olid genereeritud võtmed 750 000 Eesti ID-kaardi jaoks. Selle magistritöö raames implementeeriti ROCA rünnak ning genereeritud võtmeid ja haavatavaid kiipkaarte analüüsides loodi rünnakust uus, optimiseeritud versioon, mille abil on võimalik sooritada rünnak 140.8 aasta asemel 35.2 CPU aastaga 90% võtmete puhul ning 70.4 aastaga ülejäänud võtmetel. Lisaks loodi paralleliseeritud versioon rünnakust kasutades teadusarvutuste klastrit (HPC).

**Võtmesõnad:** Krüptograafia, kiipkaart, ROCA, RSA, Coppersmith, Faktoriseerimine, Entroopia, Küberturvalisus, Kõrgemahulised arvutused

**CERCS:** P170, Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

# Contents

# 1 Introduction

In October 2017, Czech researchers Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas published the paper "The return of Coppersmith's attack: Practical factorization of widely used rsa moduli" [1], where they described the vulnerability CVE-2017-15361 (the ROCA attack) in Infineon's proprietary RSA key generation algorithm. The researchers found that 2048-bit RSA keys can be factored in only 140.8 CPU-years. A number of widely used products, such as BitLocker TPM, YubiKey 4 and Chrome OS (many Chromebooks (Lenovo, Acer, HP, etc.)), were affected and had to find their own solution for mitigation. All of these products relied on the vulnerable Infineon RSA firmware $\leq$ 1.02.013 [2]. In response to this vulnerability, Infineon updated the library.

The flaw affected 750 000 Estonian ID cards, which used 2048-bit RSA keys generated by Infineon's vulnerable key generation algorithm. All Estonian citizens are given an official ID card containing a security chip with a RSA private key, which can be used in a variety of public services ranging from e-voting to medical prescriptions. As the cards are central to the information systems in Estonia, offering official services to their citizens and considering electronic signatures legally binding, the vulnerability had a consequential impact on Estonia. The Estonian ID cards were updated to abandon the RSA algorithm and use elliptic curve cryptography, also offered by the library on these smartcards.

The main contribution of this thesis is the implementation of the ROCA attack, which has has not been publicly available up to now. The second main contribution of this thesis is the optimization of the attack, based on the properties observed from the keys generated by Infineon's vulnerable algorithm. The analysis focused on the keys generated by the specific smartcard Infineon JavaCard SLJ52GCA150 [3], which is from the same family of security chips as used by the affected Estonian ID card platform. The optimizations found in this work are likely to apply also for other affected products from Infineon, but this has not been verified. As additional contribution, a parallelized version of the attack that can be executed on an HPC is provided.

# 2 Background and Prerequisites

This section introduces the RSA cryptosystem [4], as well as the prime generation algorithm, whose insecure implementation by Infineon led to the ROCA vulnerability. The LLL algorithm [5] and Coppersmith's attack [6], needed to implement the ROCA attack, are also presented.

## 2.1 Rivest-Shamir-Adleman (RSA) cryptosystem

Rivest-Shamir-Adleman (RSA) [4] is a public-key cryptosystem. The idea is to separate the encryption and decryption process by differentiating the key for encryption and decryption. The encryption key is called the public key and is known by everyone and the decryption key is called the private key and is only known to the key owner. The system works as follows:

**Algorithm 1:** RSA Key Generation

**Input:** keysize
**Result:** private key: $(e, d, N, p, q)$, public key: $(e, N)$

$\mathbf{p}, \mathbf{q} \xleftarrow{\$} \{\textbf{Prime numbers} > \frac{\textbf{keysize}}{\mathbf{2}}\}$
$N \leftarrow p \cdot q$
$\varphi(N) \leftarrow (p-1) \cdot (q-1)$
$e \xleftarrow{\$} \mathbb{Z} : e < \varphi(N),\ gcd(e, \varphi(N)) == 1$
$d \leftarrow e^{-1} (\text{mod } \varphi(N))$
return $(e, d, N, p, q), (e, N)$

**Algorithm 2:** RSA Encryption

**Input:** public key: $(e, N)$, message: $m$
**Result:** ciphertext: $c$

$c = m^e (\text{mod } N)$
return $c$

**Algorithm 3:** RSA Decryption

**Input:** private key: $(e, d, N, p, q)$, ciphertext: $c$
**Result:** message: $m$

$m = c^d (\text{mod } N)$
return $m$

Note that this system is the pure mathematical version of RSA, and does not include padding schemes, or further constructions. In practice, padding schemes, such as PKCSv1.5 [7], are used. There is also the possibility to use the Chinese Remainder Theorem (CRT) [8] to speedup the decryption process.

This cryptosystem was used in the Estonian ID cards, where private keys were generated on the smartcards. The key generation algorithm is the center of attention, as it is there that the ROCA vulnerability lies. The key generation algorithm needs two prime numbers $p$ and $q$. These prime numbers are supposed to be around half the key size, so that the size of their product is equal to the key size. In the key generation process above, they have an entropy of half the key size. The security assumption of RSA relies on the fact that $p$ and $q$ are selected randomly to have this entropy and on the fact that factoring big numbers is considered hard. If the prime has a lower entropy than expected, the security level of RSA will decrease, which is the case in the ROCA attack.

## 2.2 Joye and Paillier's Algorithm for Fast Prime Generation

In the original ROCA paper, it has been suggested that Infineon used a modified version of Joye and Paillier's algorithm for key generation on their security chip. This algorithm was introduced in the paper "Efficient generation of primes" [9], and its revision "Fast generation of prime numbers on portable devices: An update" [10]. The algorithm is proven secure in the context of RSA key generation and is intended for use in portables devices, such as smartcards, as the output entropy loss is negligible at practical key sizes. As explained in section 2.1, the generation of primes has to keep the security assumption strong. The problem is that generating large prime numbers is a slow process, since simply picking random odd numbers has a low probability of yielding a prime number. A secure algorithm to produce large prime numbers efficiently (high probability to be prime) is thus of high value, specifically for constricted environments. The algorithm is described below.

**Algorithm 4:** Fastprime; (based on: [10])

**Input:** $t, v, w, a \in \{\frac{\mathbb{Z}}{m\mathbb{Z}}\}^* \backslash \{1\}$
**Result:** random prime q

$\Pi = \prod_i p_i$, with $p_i$ as first i primes
$l \leftarrow v\Pi$
$m \leftarrow w\Pi$
$k \xleftarrow{\$} \{\frac{\mathbb{Z}}{m\mathbb{Z}}\}^*$
$q \leftarrow ((k - t) \bmod m) + t + l$
**while** $q$ *is not prime* **do**
    $k \leftarrow a \cdot k \pmod m$
    $q \leftarrow ((k - t) \bmod m) + t + l$
    return $q$

Infineon's modification to this algorithm is the cause of the vulnerability ex-

ploited by the ROCA attack, since the primes are not picked randomly, therefore a bias was introduced and the RSA security assumption is weakened. Infineon's modification is calculated by taking $\Pi = M$, $v$ picked randomly (corresponds to $k$ in the ROCA paper), $a = w = 1$, $t = 0$. Instead of picking $k$ (the one from Joye and Paillier, not ROCA) randomly, it is defined to be 65537 to the power of a smaller random number, modulo $M$, which is called $a$ in the ROCA paper. Since this $k$ is a random power of 65537 (mod $M$) and not a random number, the entropy is lower than it should be.

## 2.3 Lenstra–Lenstra–Lovász (LLL) Lattice Basis Reduction Algorithm

The Lenstra–Lenstra–Lovász (LLL) algorithm [5] is an algorithm for lattice base reduction. It gives the possibility to reduce a lattice base to a short and usable base in polynomial time. This algorithm is used in the Coppersmith attack explained in the next section, which constitutes a key role in the construction of the ROCA attack. LLL is widely used for factorization in different contexts, and, in this case, for breaking the security assumption of RSA. This algorithm is important in the attack, since it consumes the majority of the computing spent in the attack.

LLL is used as follows: given a basis $B = \{\{b_0, ..., b_n\}, b_i \in Polynomials_{\mathbb{Z}}\}$, a lattice is defined by taking all integer combinations of this basis: $L = \{\sum_{i=1}^{m} z_i b_i : z_i \in \mathbb{Z}, b_i \in B\}$. Given a lattice basis $B = \{\{b_0, ..., b_n\}, b_i \in Polynomials_{\mathbb{Z}}\}$, the LLL algorithm yields a new basis $B' = \{\{b'_0, ..., b'_n\}, b'_i \in Polynomials_{\mathbb{Z}}\}$ which is shorter (first polynomial $b'_0$ close to shortest vector) for the same lattice. The shortening is exponentially dependent on the number of vectors in the base. The

definition of the algorithm is:

**Algorithm 5:** Lenstra-Lenstra-Lovász (based on: [11])

**Input:** Basis: $B = \{\{b_0, ..., b_n\}, b_i \in Polynomials_{\mathbb{Z}}\}$
**Result:** Shorter basis: $B' = \{\{b'_0, ..., b'_n\}, b'_i \in Polynomials_{\mathbb{Z}}\}$

```
# Normalization is not necessary, just orthogonalization
B' = Gram_Schmidt(B)
i ← 1
while i < n do
    # Reduce
    for k ← 0 to i do
        j = (i − 1) − k
        b_i ← b_i − ((b'_ij + 1)/2) · b_j
        B' = Gram_Schmidt(B)
    if ||b_i||^2 < 2 · ||b_{i−1}||^2 then
        # Swap
        b_i, b_{i+1} ← b_{i+1}, b_i  B' = Gram_Schmidt(B)
        i ← i − 1
    else
        i ← i + 1
return B'
```

Internally, LLL uses the Gram-Schmidt transformation to construct orthogonal vectors for the new base, which is described below.

### 2.3.1 Gram-Schmidt Orthonormalization Process

Gram-Schmidt [12] is an algorithm used to transform a basis of a set to an orthonormal basis of the same set. Gram-Schmidt yields a basis which is normalized by dividing the resulting vectors by their own size, and orthogonal by recursively applying projectors from each vector on to the next one. This is always possible as all vectors in the basis are by definition linearly independent. The algorithm works as follows:

**Algorithm 6:** Gram-Schmidt

**Input:** Basis: $B = \{b_0, ..., b_n\}$
**Result:** Orthonormal basis: $B_{GS} = \{b'_0, ..., b'_n\}$

**Orthogonalize:**
$b'_0 = b_0$
$b'_1 = b_0 - proj_{b_0}(b_1)$
$b'_2 = b_0 - proj_{b_0}(b_2) - proj_{b_1}(b_2)$
...
$b'_n = b_0 - proj_{b_0}(b_n) - ... - proj_{b_{n-1}}(b_n)$
**Normalize:**
return $\{\frac{b'_0}{||b'_0||}, ..., \frac{b'_n}{||b'_n||}\}$

The algorithm defines projectors from each vector on the next one so that the resulting vectors $b'_i$ will be orthogonal to all other vectors in the new basis. As an example, in a two dimensional space the basis $B = \{b_0, b_1\}$, Gram-Schmidt constructs a right-angled triangle with the vector $b_0$ and an orthogonal vector $b_0^{\perp}$ to construct $b_1$. Thus, $b'_0 = b_0$ and $b'_1 = b_0 - \underbrace{proj_{b'_0}(b_1)}_{b_0^{\perp}}$. The new basis can then be constructed with $B_{GS} = \{\frac{b'_0}{||b'_0||}, \frac{b'_1}{||b'_1||}\}$.

## 2.4 Coppersmith's Attack

Don Coppersmith, in [6], presented an attack on the RSA cryptosystem. The attack allows to decrypt a message encrypted with RSA, if enough bits from the original message are known. The attack is done in seven steps, where the ciphertext to be decrypted is transformed. There have been other optimizations for other purposes, such as the extension by Boneh and Durfee [13], who found an improvement where the secret key can be revealed (in contrast to just decrypting one message), which is a complete break of the cryptosystem with partially known plaintext. The particular form of Coppersmith's used in the ROCA attack is the Howgrave-Graham extension [14]. This extension is used in each iteration of the main attack and is used to factor a potential candidate of $p$ or $q$. The attack is described below.

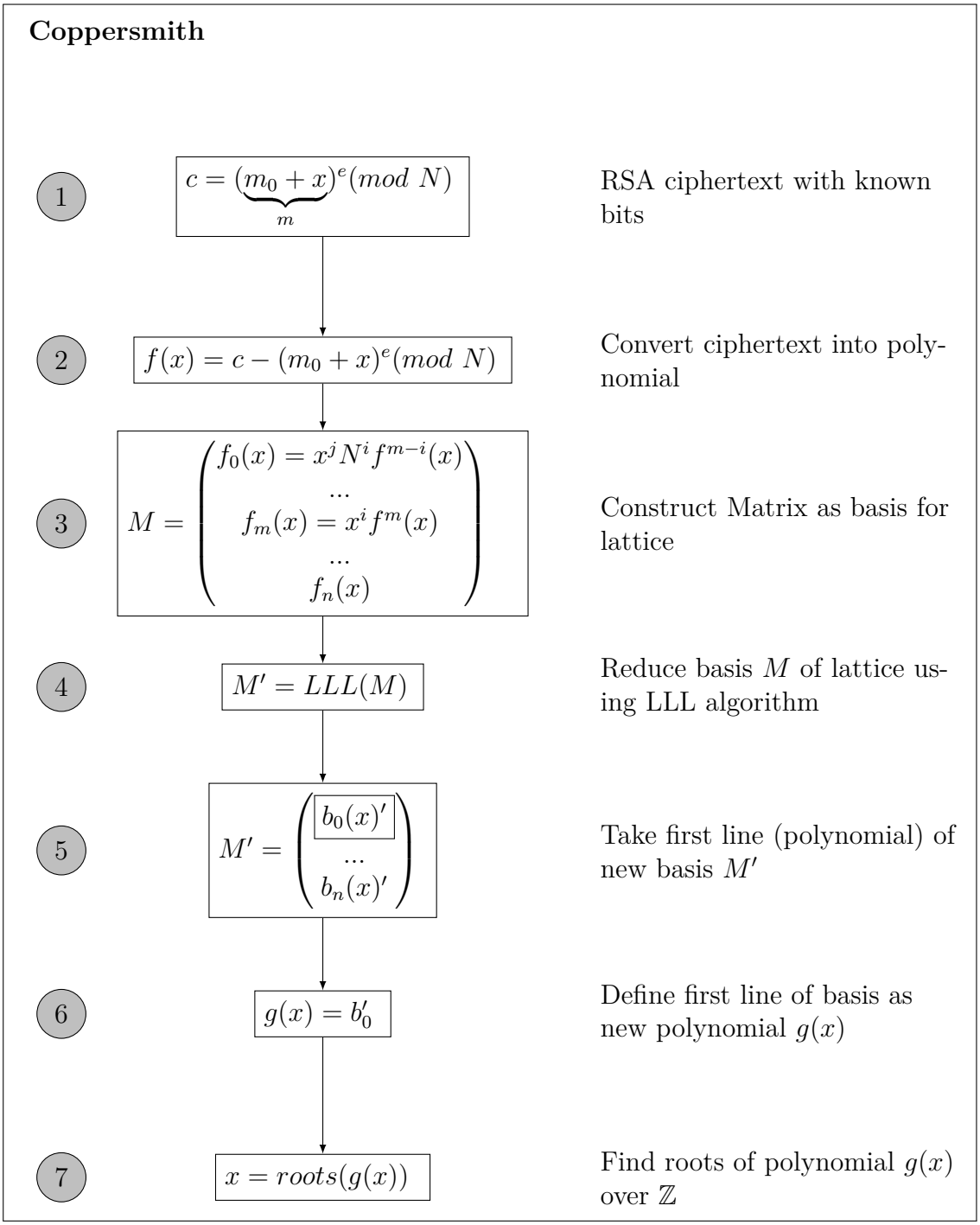


Figure 1: Overview of the Coppersmith attack

### 2.4.1 Transform Ciphertext into a Polynomial

The first step of Coppersmith's attack requires to know some bits of the ciphertext to be decrypted. The second step in this attack is to transform the known ciphertext into a polynomial. In the case of ROCA, the polynomial is not taken from the ciphertext, but constructed from the structure of the key. In this example, the partially known plaintext will be used. As some bits of the encrypted message are known the message can be separated into two parts, namely the known bits and the unknown bits. The encryption key is not important, as it does not take any part in the attack. A RSA encrypted message $m$ is of the form $c = m^e (mod\ N)$. Since part of the message $m$ is known, it is possible to cut $m = \underbrace{m_0}_{known} + \underbrace{x}_{unknown}$, which would look something like

$$\text{“}\underbrace{\text{The password is: }}_{known}\underbrace{\text{mysecretpass123}}_{unknown}\text{”}$$

or

$$\text{“}\underbrace{\text{AAAAAAAAAAA...}}_{\text{known padding}}\underbrace{\text{mysecretpass123}}_{unknown}\text{”}$$

The minimal condition for Coppersmith to work is $|x| < N^{\frac{1}{e}}$. The form of the polynomial does not matter for Howgrave-Graham as long as it is univariate (one unknown variable). This is the case in the ROCA attack, where the polynomial is different, but also univariate. Note that there exists a multivariate version of Howgrave-Graham which is not used in ROCA. It is possible to transform the ciphertext into a polynomial of the form:
$f(x) = c - (m_0 + x)^e (mod\ N)$, by just passing c to the other side of the equation and substituting $m$ by $m_0 + x$. The idea is to find the roots of this polynomial, as they describe the value of the unknown $x$, which is the unknown part of the ciphertext. Since this is not a trivial problem, other steps are needed to transform this polynomial.

### 2.4.2 Find Equivalent Polynomial in $\mathbb{Z}$

Since the polynomial which is transformed from the ciphertext is modulo $N$, it is necessary to find an equivalent polynomial in $\mathbb{Z}$, with the same roots. Finding the roots of a polynomial over $\mathbb{Z}$ is easy, in contrast to finding roots of polynomial over a ring. Howgrave-Graham shows in his paper that under certain conditions, another polynomial $g(x)$, over $\mathbb{Z}$, which has the same roots as our polynomial $f(x)$, can be found with the following theorem:

***Theorem 1***
*Let* $g(x_0) \equiv 0 \ mod \ N^m$ *with degree* $d-1$ *and* $|x_0| < X \ ||g(xX)|| < \frac{N^m}{\sqrt{\delta}} \Rightarrow \exists g(x_0) = 0$ *over* $\mathbb{Z}$

Source: [14]

### 2.4.3 Lattice Reduction

Finding an equivalent polynomial over $\mathbb{Z}$ is not enough as it also has to be small enough to be factored (its roots being found). It is needed to construct a set of polynomials used to construct the final polynomial $g(x_0)$, which can be factored. This construction (step 3 in Figure 1) will be used to find the suitable polynomial to be factored. The polynomials are fed into a lattice, so that the reduction (step 4 in Figure 1) of the lattice yields smaller polynomials which are suitable. The polynomial $g(x_0) < p^m$ is constructed by using a set of polynomials $f_i$ of the form:

**From** $0$ **to** $m-1$**:** $f_i(x) = x^j N^i f^{m-i}(x) \ i = 0, ..., m-1 \ j = 0, ..., \delta - 1$

**from** $m$ **to** $t-1$**:** $f_{i+m}(x) = x^i f^m(x) \ i = 0, ..., t-1,$

where $\delta$ is the degree of $f(x)$. The $t$ and $m$ parameters are given as optimization parameters depending on the specific polynomial $f(x)$. This is used to construct a basis matrix of a lattice. When reduced, the first polynomial $b'_0$ of this lattice is defined as being our polynomial: $g(x_0)$ (step 5 and 6 in Figure 1), which is evaluated with $g(xX)$ with $X = \lceil N^{\beta^d - \epsilon} \rceil$. In order to reduce the lattice, the LLL algorithm is used (see Section 2.3). When the new polynomial $g(x_0)$ is known, its root are taken with standard methods in $\mathbb{Z}$ (step 7 in Figure 1). Finally, one of the found roots is the secret message $x$. The variables $c, m_0, e, N$ are known ($(e, N)$ is the public key of RSA) and $x$ is the variable to be found. As $x$ is known, the full ciphertext is decrypted and RSA is broken in this context.

# 3 The ROCA Attack

This section explains the original attack by the Czech researchers. The ROCA attack takes advantage of the Coppersmith attack, or more precisely, the Howgrave-Graham extension of it. It takes advantage of a found polynomial form of the primes to introduce a Coppersmith-based bruteforce on a reasonably-sized parameter range. As stated in the ROCA paper, it was found that in Infineon's version of the RSA key generation, the primes used for the private key were constructed using a specific polynomial form. This polynomial form is a variant of Joye and Paillier's algorithm (see Section 2.2), which, in itself, is a secure prime generation

algorithm, contrary to the version in the vulnerable implementation by Infineon. In Infineon's key generation algorithm, the primes $p$ and $q$ are generated using the polynomial [1]:

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

where $M$ is known, but $a$ and $k$ are random integers. This form reduces the searching space from a random prime half the key size, to the size of the unknown $k$ and $a$. In the case of $k$ and $a$, the search space for $a$ still is too big for a reasonable bruteforce. The parameter $M$ is a known primoral constant, meaning that $M$ is the product of all primes up to $n$: $M = \prod_1^n primes$, where $n$ depends on the key size see parameter $n$ in Table 2). The form is chosen specifically so that $p$ will not contain the divisors in $M$ and thus have more chance of being a prime. For example a key of 512 bits results in an effective entropy of 99 bits, as the size of $a$ and $k$ together is 99 bits: $99 = 62 + 37 = log_2(a) + log_2(k)$. Since $M$ is a product of primes, it is possible to move some entropy from $a$ to $k$, by increasing the size of $k$ with some divisors of $M$. This will make the order of $k$ smaller, therefore $a$ will also be smaller. As the range of $a$ becomes smaller, bruteforcing will become possible. For example, $2^{62}$ is considered too big to bruteforce, since each iteration is time consuming because of the LLL reduction. It is thus important to find a small enough range for $a$ to be bruteforcable and a small enough $k$ to be calculable by the LLL reduction and root finding of the polynomial. The transformation is made by transferring divisors (primes) of $M$ to $k$. The idea of the ROCA paper is thus to find a new $M', k', a'$ which are reasonable parameters for the bruteforce. Since both primes are of this form, the public key is of the form

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

Source: [1]
The ROCA attack is depicted in Figure 2.

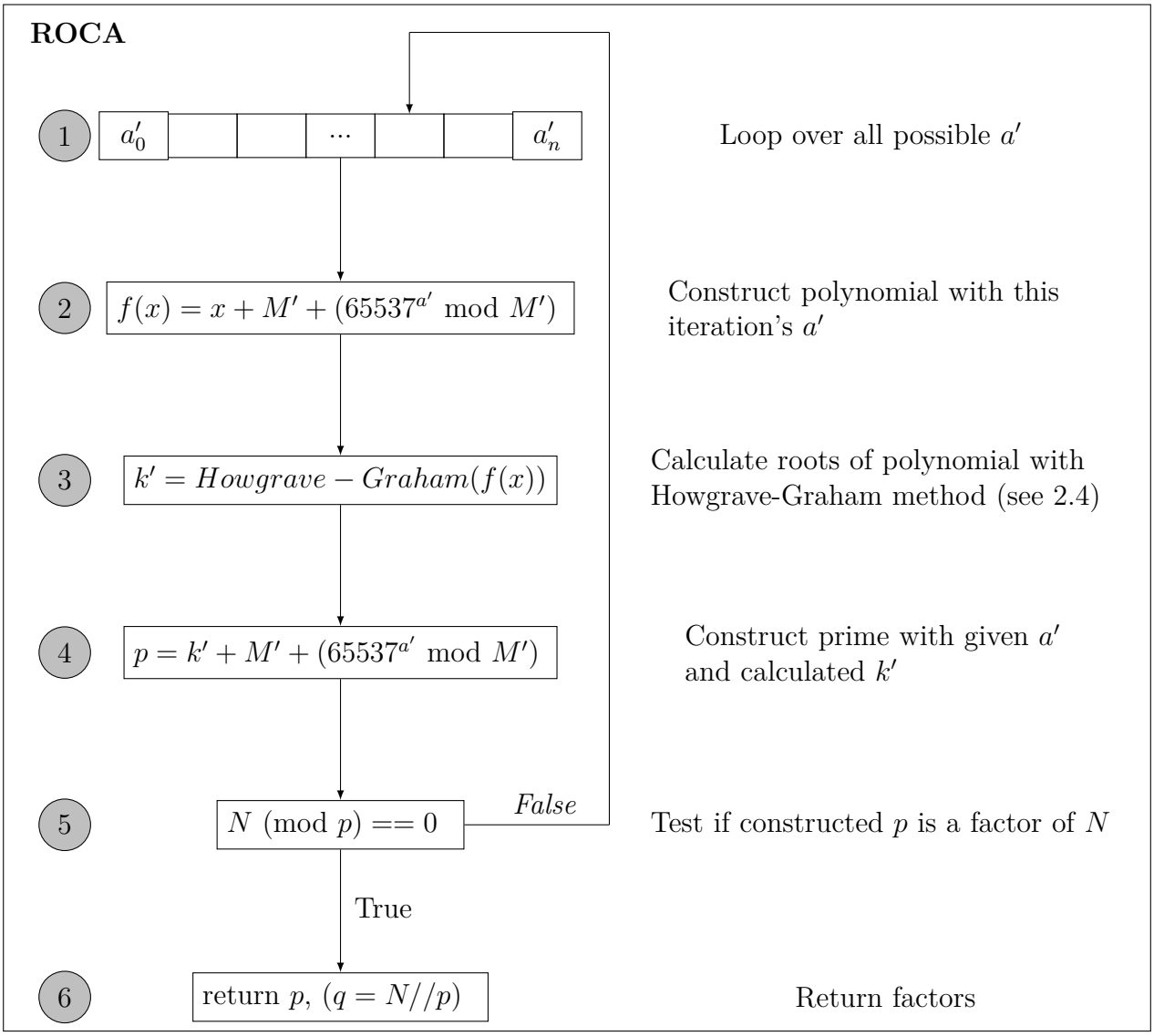


Figure 2: Overview of the ROCA attack

**Algorithm 7:** ROCA

**Input:** N
**Result:** p,q: factors of N

**foreach** $a'$ **do**
  $f(x) = x \cdot M' + (65537^{a'} \bmod M')$
  $k' = \text{coppersmith}(f(x))$
  $p = k' \cdot M' + (65537^{a'} \bmod M')$
  **if** $(N \ mod \ p) == 0$ **then**
    return $p$, $\frac{N}{p}$

This attack is possible as the primes are in this polynomial form (see equation (1)). The Czech researchers reverse-engineered the form by taking small primes as modulo of the public key and plotting them to see if the occurrence is the same for all of them or if some moduli are more common than others. It was found that in Infineon smartcards, certain moduli show a strong bias.

# 4 Optimization Based on Properties Observed from Real Keys

This section focuses on finding further improvements for the ROCA attack, based on properties observed from real keys. In the original paper, as a potential improvement, the authors suggested looking into the polynomial choices for constructing the lattice to be factored. If it is possible to find a more suitable polynomial for the lattice construction, the attack could greatly benefit from it. As the original construction could not be optimized further, the variables in the construction of the polynomial (1) were analyzed. If the entropy of the parameters used to construct the primes is lower than expected, the bias can be used as an advantage in the attack. The original paper considers integers $a$ and $k$, used to construct primes, to be random, however, from actual keys, generated by the vulnerable key generation algorithm by Infineon, it can be seen that they have biases which can be exploited to optimize the attack further.

In order to confirm the findings and be fully compatible with the original paper, as well as confirm that the Estonian ID cards are affected, the private key harvesting has been done with the Infineon JavaCard SLJ52GCA150 [3]. Generating private keys from real cards can give unique insight to make multiple discoveries. First of all, confirm the findings of the ROCA paper by testing the polynomial form of the primes. Secondly, confirming that the implementation of the attack works on real keys (see section 5). Finally, finding potential new flaws by analyzing the entropy of these keys. For extracting the keys, the JavaCard

private key extracting tool and its client from the Czech researchers in [15] has been used to ensure compatibility and correctness. A custom python script has been written to parse the output of this tool and to analyze the private keys. An array of 9 cards has been used to harvest 1 250 000 private keys (i.e, 2 500 000 primes). The focus of this work is on 2048-bit RSA keys, since they are widely used, including by the Estonian ID cards before migration to ECC. The same properties have also been observed on RSA 512-bit and 1024-bit keys extracted form the Infineon JavaCard SLJ52GCA150. The parameters $a$ and $k$ were recovered from the primes $p$ and $q$ using the formula $a = discrete\_log_{65537}(p \bmod M)$, $k = \frac{p-(p \bmod M)}{M}$ (to compute $a'$ and $k'$ the same formula is used by substituting $M$ with $M'$). Since the order of the group is small, the discrete logarithm can be computed. The parameter $c$ (or $c'$) can be calculated from the public key, using the formula $c = discrete\_log_{65537}(N \bmod M)$.

## 4.1 Fixed Bits in Exponent $a$ and $k$

In the original paper integers $a$ and $k$, as well as $a'$ and $k'$, were considered to be uniformly random. The entropy of the parameters $a'$ and $k'$ of the modified prime generation polynomial $p = k' * M' + (65537^{a'} \bmod M')$ has been analyzed. The parameters $a'$ and $k'$ are similar to $a$ and $k$, but using $M'$ instead of $M$. In Figure 3, the probability distribution of each bit in $a'$ and $k'$ is shown. The expected result would be the same probability between 1 and 0, represented by the dotted blue line. The integers $a$ and $k$ are of the same form as $a'$ and $k'$, since $M'$ and $k'$ are calculated by transferring some divisors of $M$ to $k$ (see [1]). Therefore, any bias in $a$ and $k$ will also be in $a'$ and $k'$.

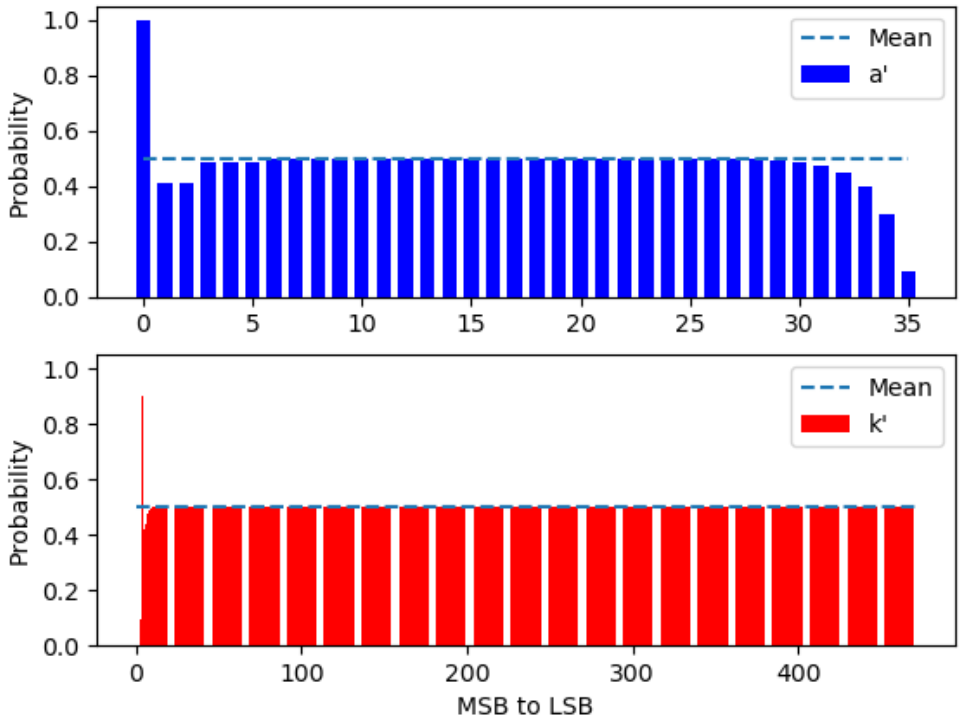


Figure 3: Entropy of each bit in $a'$ and $k'$, MSB to LSB (2048-bit keys)

As can be seen in Figure 3, some bits are strongly biased or even fixed at a particular value. For example, the parameter $a'$ has its most significant bit (MSB) fixed to 1. This means that all values below $a' = 2^{\text{bitlen(a')}-1}$ never occur and can be excluded from the bruteforce range of $a'$. The new value of the optimized bruteforce range is defined by $c_a = 2^{\text{bitlen(a')}-1}$, as shown in Figure 4.

The optimization gained by bruteforcing from $c_a$ can be calculated by the difference between $[\frac{c'}{2}, \frac{c'+ord'}{2}]$ and $[c_a, \frac{c'+ord'}{2}]$. Since $ord' >> c'$, the speedup of the optimization will have a small variance. Since $c_a$ is the half of the theoretical maximum, the worst-case speedup is twice faster. To summarize, the optimization will cut the range by two plus the trivial variance. Theoretically, this cuts down the worst case time required to factor 2048-bit RSA key from 140.8 CPU-years of the original attack to 70.4 CPU-years.

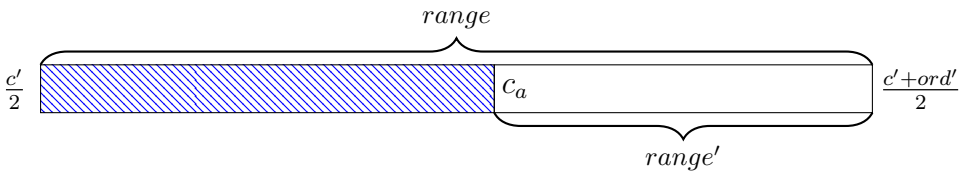


Figure 4: Comparison of the original and the new bruteforce range

The second observation is that the parameter $k'$ is fixed at the 3 MSBs. This can be used to transform the polynomial to the form

$$p = \overbrace{c_k + r}^{k'} * M' + (65537^{a'} \bmod M') \tag{3}$$

where $c_k$ is known. This transformation can be used by passing $c_k$ to the side of $p$ and reconstructing $p - c_k$ instead. As this does not speedup the attack in practice, it is currently not used. However, it is possible that another transformation could speedup the root-finding algorithm, since some bits of the root are already known, namely $c_k$ (3 MSBs of $k'$).

## 4.2 Biases in Exponent $a'$

In addition to the fixed MSB in $a'$, there are other bits in $a'$ which are biased. When looking at the LSB, one can notice that it is biased towards 0. It means that $a'$ is statistically biased towards being even and this can be used to speedup the attack. The optimized attack thus iterates first over even $a'$ and then over odd $a'$. For 2048-bit RSA keys, the LSB of $a'$ is even 90.98% of the time (for 1024-bit: 95.7%, 512-bit: 97.2%). The average gain of iterating over even $a'$ first, is a speedup of $\frac{range}{2} \cdot 0.9098$. Theoretically, this cuts down the worst case time required to factor 2048-bit RSA key from the previously improved 70.4 CPU-years to 38.7 CPU-years for a random key.

## 4.3 Cherry-picking Weaker Public Keys

Since the range of the bruteforce depends from $c'$ which can be extracted from the public key, a key with a small $c'$ can be cherry-picked for a smaller bruteforce range. The RSA public keys from the Estonian ID cards certificates, affected by the ROCA vulnerability, have been searched for the smallest $c'$. In total 1 758 745 affected Estonian ID cards certificates were analyzed (this number includes renewed certificates). The smallest $c'$ found was $c' = 20619$. Since the range goes from $\frac{c'}{2}$ to $\frac{c'+ord'}{2}$, a smaller $c'$ will have a smaller range to iterate over, and will thus take less time. This speedup does not make a big difference, because $ord' >> c'$

(order is much bigger than $c'$). On a 2048-bit key, it speeds up the attack by around 2%.

Another possibility is to cherry-pick an odd $c'$, as the only possibility to have an odd $c'$ is having an even $a'$ and odd $b'$ or the opposite (since the sum of any even and odd number is odd). Thus by iterating only over the even numbers, $a'$ will always be found. This represents an average speedup of 90.98% in general, as now the range is cut in half again, and 100% if $c'$ is odd. In comparison to the original attack, the range now is four times smaller with a cherry-picked key and the previous optimizations. This cherry-picking is thus guaranteed on 90.98% of the keys. Theoretically, this cuts down the worst case time required to factor 2048-bit RSA key from the previous 70.4 CPU-years to 35.2 CPU-years.

## 4.4 Efficiency of the Optimized Attack

The theoretical efficiency of the optimized algorithm using the previously described optimizations, is now in the worst case, using only the biased bits in $a'$,
$efficiency_{worstcase} = \frac{|\{range_{a'}\}|}{2}$.

In the worst case using a random key and exploiting the biased bits in $a'$,
$efficiency_{worstcase,random\ key} = \frac{|\{range_{a'}\}|}{4} \cdot 0.9098(bias)$

and cherry-picked and using the biased bits
$efficiency_{worstcase,cherry-picked\ key} = \frac{|\{range_{a'}\}|}{4}$.

The Table 1 shows the worst case cost of the attack using the ROCA implementation developed in this work, executed on the university cluster. The HPC of the University of Tartu uses a slightly slower CPU than the researchers used in the original paper. The HPC uses the CPU "Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz" with an Intel Thermal Design Power (TDP) of 95W for 20 cores.

The table is divided by key size and by the type of the attack used. The numbers marked with a "*" are actual running times, the other values are extrapolated from the duration of one Coppersmith iteration and calculated using the worst case efficiency range.

The price of the calculation is provided based on the energy consumption specification of the CPU used by the HPC and the energy price of 0.2$/kWh (the same as used in the original paper).

| **Key size** | *Non-optimized* | *Optimized* | *Optimized Random key* | *Optimized Cherry-picked* |
|---|---|---|---|---|
| **512-bit** | 2.0333 CPU-hours* | 2.2 CPU-hours | 0.73 CPU-hours* | 0.51 CPU-hours* |
| **1024-bit** | 102.4 CPU-days | 51.2 CPU-days | 36.5 CPU-days | 25.6 CPU-days |
| **2048-bit** | 161.2 CPU-years | 80.6 CPU-years | 57.5 CPU-years | 40.3 CPU-years (336$) |

Table 1: Efficiency of the ROCA attack using HPC

# 5 Implementing the ROCA Attack

The attack has been implemented using the description provided in the ROCA paper. In the original paper, an example for the $M$ to $M'$ transformation is given, but includes an error, which created difficulties for the implementation and validation, since the parameters were not publicly available. The example 2.1 in Section 2.7.2 of the original paper, missed the factor 53 in its power prime divisors. In the best candidates for $p^{e_j}$, the value $3^3$ should be used instead of the value $3^2$ as provided in the paper. An discussion with the authors of the original paper solved the issue and confirmed the obtained parameters.

As a further contribution, some of the parameters, needing a lot of computing, have been pre-calculated and are shown in Table 2. The parameter $n$ is the number of consecutive primes to be multiplied to get $M$. Parameters $a_{max}$ and $k_{max}$ are the number of bits of $a$ and $k$. $M$ is the product of n primes. $M'$ is the optimal parameter for the attack, derived from $M$ by transferring divisors to $k$. Parameters $m$ and $t$ are optimal parameters for the underlying Coppersmith's method given in the ROCA paper. Finally, $c_a$ is the optimization parameter defined in Section 4.1.

| | **512** | **1024** | **2048** |
|---|---|---|---|
| $n$ | 39 | 71 | 126 |
| $a_{max}$ | 62 | 134 | 434 |
| $k_{max}$ | 37 | 37 | 53 |
| $M(Hex)$ | 09 24 cb a6 ae 99 df a0 84 53 7f ac c5 49 48 df 0c 23 da 04 4d 8c ab e0 ed d7 5b c6 | 07 92 3b a2 5d 12 63 23 28 12 ac 93 0e 96 83 ac 0b 02 18 0c 32 ba e1 d7 7a a9 50 c4 a1 8a 4e 66 0d b8 cc 90 38 4a 39 49 40 59 34 08 f1 92 de 1a 05 e1 b6 16 73 ac 49 94 16 08 83 82 | 07 cd a7 9f 57 f6 0a 9b 65 47 80 52 f3 83 ad 7d ad b7 14 b4 f4 ac 06 99 97 c7 ff 23 d3 4d 07 5f ca 08 fd f2 0f 95 fb c5 f0 a9 81 d6 5c 3a 3e e7 ff 74 d7 69 da 52 e9 48 d6 b0 27 0d d7 36 ef 61 fa 99 a5 4f 80 fb 22 09 1b 05 58 85 dc 22 b9 f1 75 62 77 8d fb 2a ea c8 7f 51 de 33 9f 71 73 1d 20 7c 0a f3 24 4d 35 12 9f eb a0 28 a4 84 02 24 7f 4b a1 d2 b6 d0 75 5b af f6 |
| $M'(Hex)$ | 1b 3e 6c 94 33 a7 73 5f a5 fc 47 9f fe 40 27 e1 3b ea | 24 68 31 44 f4 11 88 c2 b1 d6 a2 17 f8 1f 12 88 8e 4e 65 13 c4 3f 3f 60 e7 2a f8 bd 97 28 80 74 83 42 5d 1e | 01 69 28 dc 3e 47 b4 4d af 28 9a 60 e8 0e 1f c6 bd 76 48 d7 ef 60 d1 89 0f 3e 0a 94 55 ef e0 ab db 7a 74 81 31 41 3c eb d2 e3 6a 76 a3 55 c1 b6 64 be 46 2e 11 5a c3 30 f9 c1 33 44 f8 f3 d1 03 4a 02 c2 33 96 e6 |
| $m$ | 5 | 4 | 7 |
| $t$ | 6 | 5 | 8 |
| $c_a(Hex)$ | 08 00 00 | 40 00 00 00 | 04 00 00 00 00 |

Table 2: Optimal parameters for the ROCA attack

The attack code is written in python 2.7 using SageMath library [16]. For Coppersmith's attack, the already existing SageMath implementation of Howgrave-Graham [17] was used (the same as used by the Czech researchers). For the LLL implementation, SageMath uses the fpyLLL wrapper from the fpLLL library [18]. The code has been separated in multiple modules. The code which calculates the parameters given in Table 2, can be found in the "param.py" file. The attack is implemented in "roca.py". A pure SageMath version is also included in "roca.sage". In order to confirm that the attack works, a test case can be found in the "test" folder. The test case runs the attack by calculating $k'$ with known $a'$ and finds $p$ by reconstructing the polynomial. The "data" folder contains multiple vulnerable keys with different sizes, which can be used for testing purposes. The code takes as an input the public keys accepted in the format supported by pycrypto [19], namely X.509 [20] in binary or PEM, PKCS#1 in binary or PEM and OpenSSH [21] public key.

To run the original attack (without the optimizations found in this work), the number of cores to be used (default: 1) can be passed to the implementation and used with the following command:

```
$ python2 roca.py <path to key> -j <number of cores>
```

The attack prints out the primes and exports the private key to "priv.pem" file. The Listing 1 shows the output from the execution of the attack against a vulnerable 512-bit RSA key.

```
$ python2 roca.py data/512.pem
[+] Importing key
[+] Key is vulnerable!
[+] RSA-512 key
[+] N = 80474497870208039394761476993787283293147334261
64267535316072793294233587337682475529099270039635820
02260707371017160997944821548814875889400167842361138 9
[+] c' = 588970
[+] Time for 1 coppersmith iteration: 0.04 seconds
[+] Estimated (worst case) time needed for the attack:
4 hours, 30 minutes and 3.46 seconds
[+] Found factors of N:
[+] p = 89316585341239200103164798629168230185983346548 5
58222489515734210664382833579
[+] q = 90100284916570139638439028489735281413386074204 4
03670730318637933879173958391
[+] Took 7742.3 s
[+] Exporting key to priv.pem
```

Listing 1: Execution of the attack against vulnerable key

The optimized version of the attack (see Section 4) is used with the same parameters and can be found in "optimization.py". This implementation of the attack applies all the optimizations presented in this thesis. It will iterate only from $c_a$ and first over even numbers. If the key is cherry-picked, namely that $c'$ is odd, this will be detected and the time estimate will be updated.

To the best of our knowledge, this implementation is the only complete implementation of the attack, which is publicly available. The code has been published on GitHub [22]. Until now, the only available code for the attack was done by Bernstein & Lange [23], and requires the knowledge of two parameters $u$ and $v$ for the key (corresponds to $a$ and $k$). The project CheckResearch reproduced the paper [24] using the script from Bernstein & Lange to confirm the findings.

## 5.1 Parallelizing the Attack Code for HPC

Factoring a real key (bigger than 512 bits) cannot be done on a simple laptop, as it requires a lot of computing resources. Thus, it is necessary to implement a parallelized version of the attack, which can be run on a High Performance Cluster (HPC). The available HPC at the University of Tartu is based on SLURM (Simple

Linux Utility for Resource Management) [25]. A wrapping script for SLURM is included in the code base. The parallelized version is formatted to be runnable by the script or with GNU "parallel". To run the attack on SLURM the following command is used:

```
$ sbatch slurm.sh <path to key>
```

For the "roca.py" and "optimization.py" to be also usable in parallel, the non-SLURM attack is coded in a master-slave worker pool. To avoid the python Global Interpreter Lock (GIL) and have true parallelization, the code uses multiprocessing [26] instead of threading [27].

# 6 Conclusion

In the original paper, the real world factorization of vulnerable 2048-bit RSA keys was already feasible in a reasonable time and cost. With the optimizations provided in this work, the time and cost are reduced even further. The worst case time for the attack was cut down from 140.8 CPU-years to 35.2 CPU-years for 90% of the keys and 70.4 CPU-years for the remaining 10% of the keys.

Further improvements are still possible, since, as can be seen in Figure 3, some bits of $k'$ are biased. The entropy is lower than expected, thus it could be used to speed up the root finding. Another improvement could be to use other biases in $a'$ and change the iteration order again. Optimizing the LLL parameters and finding other polynomial constructions for the lattice basis still are the best candidates for further optimization. As for the total time spent on the attack, an implementation of LLL on FPGA in languages such as VHDL or Verilog would make a significant improvement. If implemented, they could speedup each iteration time and could be used by cloud services such as Amazon's F1. Other heuristics for the implementation could also help the code get faster, as the code is based on python. Usage of other languages than python could also help, although the dependency for SageMath has to be removed for this purpose.

**Acknowledgments.** This research has been carried out with financial support from the Estonian Ministry of Economic Affairs and Communications.

# List of Figures

# List of Tables



# List of Algorithms



# List of Listings



# References

[1] Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas. The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1631–1648, New York, NY, USA, 2017. ACM.

[2] NIST National Vulnerability Database. CVE-2017-15361. https://nvd.nist.gov/vuln/detail/CVE-2017-15361, October 2017.

[3] Infineon. SLJ52GCA150 JavaCard. https://secure.smartcardsource.com/slj52gdl150cl-java-smart-card.html, May 2019.

[4] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 1978.

[5] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 1982.

[6] Don Coppersmith. Finding a Small Root of a Bivariate Integer Equation; Factoring with High Bits Known. In Ueli Maurer, editor, *Advances in Cryptology — EUROCRYPT '96*, pages 178–189, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[7] IETF. PKCSv1: RSA Encryption Version 1.5. https://tools.ietf.org/html/rfc2313, May 2019.

[8] Wikipedia contributors. Chinese remainder theorem. https://en.wikipedia.org/wiki/Chinese_remainder_theorem, May 2019.

[9] Marc Joye, Pascal Paillier, and Serge Vaudenay. Efficient Generation of Prime Numbers. 08 2000.

[10] Marc Joye and Pascal Paillier. Fast generation of prime numbers on portable devices: An update. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, pages 160–173, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[11] Jeffrey De Fauw. LLL factorization. https://github.com/JeffreyDF/LLL_factorization/blob/master/factorization.py.

[12] Wikipedia contributors. Gram-Schmidt process. https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process, May 2019.

[13] Dan Boneh and Glenn Durfee. Cryptanalysis of RSA with private key d less than n0:292. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1999.

[14] Nick Howgrave-Graham. Computational mathematics inspired by RSA. 1998.

[15] Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas. JCAlgTest. https://github.com/crocs-muni/JCAlgTest, May 2019.

[16] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 8.7)*, May 2019. `https://www.sagemath.org`.

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

[18] The FPLLL development team. fpLLL, a lattice reduction library. https://github.com/fplll/fplll, May 2019.

[19] Darsey Litzenberger. Python cryptography toolkit (pycrypto). https://github.com/dlitz/pycrypto, May 2019.

[20] IETF. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. https://tools.ietf.org/html/rfc5280, May 2019.

[21] OpenBSD project. OpenSSH. https://www.openssh.com/, May 2019.

[22] Bruno Produit. Implementation of the ROCA attack. https://github.com/brunoproduit/roca, May 2019.

[23] Daniel J. Bernstein and Tanja Lange. Reconstructing ROCA. https://blog.cr.yp.to/20171105-infineon.html, 2017.

[24] Reproduction of: The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli (ACM CCS 2017). https://checkresearch.org/Experiment/View/a2326b3a-1426-4217-98ff-8a2eccbbfaba, 2018.

[25] Simple Linux Utility for Resource Management (SLURM). https://slurm.schedmd.com/publications.html, May 2019.

[26] Python contributors. multiprocessing - Process-based "threading" interface. https://docs.python.org/2/library/multiprocessing.html, May 2019.

[27] Python contributors. threading - Higher-level threading interface for python. https://docs.python.org/2/library/threading.html, May 2019.

# License

## Non-exclusive licence to reproduce thesis and make thesis public

I, **Bruno Produit**,

1. herewith grant the University of Tartu a free permit (non-exclusive license) to:

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   **Optimization of the ROCA (CVE-2017-15361) Attack**

   supervised by Arnis Paršovs, MSc

2. I grant the University of Tartu a permit to make the work specified in p.1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p.1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Bruno Produit
17.05.2019