UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Algorithms for RSA key recovery

Algorithmen zur RSA Schlüsselwiederherstellung

Bachelorarbeit

im Rahmen des Studiengangs
IT-Sicherheit
der Universität zu Lübeck

vorgelegt von
Christopher Krebs

ausgegeben und betreut von
Prof. Dr.-Ing. Thomas Eisenbarth

mit Unterstützung von
Dr. rer. nat. Sebastian Berndt

Lübeck, den 19. April 2021

Abstract

This paper deals with the RSA key recovery algorithm published by Nadia Heninger and Hovav Shacham in 2009. This algorithm is able to completely reconstruct the private key using special partial information about the private key. This information could for example be obtained using a cold boot attack. We consider the value q_p^{-1} whose information is not used in the algorithm and analyze how this information could be used to improve the algorithm.

Diese Arbeit beschäftigt sich mit dem RSA Key Recovery Algorithmus, den Nadia Heninger und Hovav Shacham 2009 veröffentlicht haben. Besagter Algorithmus ist in der Lage mithilfe von speziellen partiellen Informationen über den privaten Schlüssel, die beispielsweise über einen Cold Boot Angriff erlangt werden können, diesen Schlüssel vollständig zu rekonstruieren. Wir betrachten den Wert q_p^{-1}, dessen Informationen im Algorithmus nicht genutzt werden und analysieren wie diese Informationen genutzt werden könnten um den Algorithmus zu verbessern.

Acknowledgements

I take this opportunity to express gratitude to Sebastian Berndt for his constant supervision, help, support and feedback.
I would also like to thank my proof readers Nico Kleefeldt and Theodor Rolfs for their time and effort.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Lübeck, 19. April 2021

Contents

1 Introduction

In this thesis, we consider the key recovery algorithm for RSA private keys developed by Nadia Heninger and Hovav Shacham. The key is stored in redundant form, so that there are six values in the decryptor's memory, each of which is sufficient to completely recover the private key. In our particular case, we assume that information is available about these six values that reliably tells us the state of some bits. The algorithm can recover the complete key from bitwise known partial information about the redundantly stored private key. The algorithm uses only the first five of these six stored values. Thus, one sixth of the information obtained about the key is not used. In developing the algorithm, only five out of six available sources of information were used. Here, we consider this sixth value, investigate ways to include this value in the recovery of the private key, and show how it is possible to obtain information about the private key when this sixth value is available.

1.1 Goals

The goal of our considerations is to integrate the sixth value q_p^{-1} into the key recovery algorithm and to make the partial information obtained through it usable. We hope that this will lead to an improvement of the algorithm. On the one hand, this improvement could be an increased efficiency and thus a faster execution of the algorithm, on the other hand, it is possible to reduce the limitations of the algorithm by extending the information used. Overall, the goal is to use the obtained partial information more effectively.

1.2 Procedure

We start by reviewing the basics. To do this, we first look at the underlying encryption method RSA. We are primarily interested in the structure of the procedure. Particularly relevant for us are the parts of the key generation that must not be public. Then we look at how encryption and decryption with RSA works and why it works. Finally, we consider how difficult it is to solve the RSA problem. Next, we look at the key recovery algorithm. The main goal is to understand how and why the algorithm works. The algorithm will be put in perspective as to its capabilities and prerequisites as well as its possible uses. In principle, the algorithm is based on the fact that the information obtained about the re-

dundantly stored values of the private key is compared bit by bit in a system of equations, so that the values can be calculated relatively efficiently and reliably bit by bit starting from the least significant bit. After we are more familiar with the algorithm, we look at q_p^{-1}, which is the sixth value stored for decryption that is currently not included in the algorithm. For this purpose, we look at the computation of the value and how it is used in RSA for decryption. After that, we think about ways to find dependencies between q_p^{-1} and the other values of the private key to be able to use information about q_p^{-1}. Finally, we show how it is possible through the Coppersmith' method to compute from the value k', which gives the number of arithmetic overflows in the modular ring p such that $q \cdot q_p^{-1} - 1 = k' \cdot p$, and the public key the prime p and hence the entire key.

2 RSA Cryptosystem

RSA is an asymmetric cryptographic system developed in 1977 [RSA78], which uses integer factorization. It is the first and most widely used algorithm of this type and is valid for both encryption and digital signing. The security of this algorithm lies in a problem closely related to integer factorization. The messages sent are encoded into numbers, and the operation is based on the known product of two large random prime numbers. As in any public key system, each user has two encryption keys: one public and one private. Once someone wants to send a message they need to look up the receiver's public key, encrypt their message with that key, and once the encrypted message reaches the receiver, the receiver decrypts it using their private key. The message can be sent through a public channel, since only the holder of the private key can decrypt it within feasible time.

2.1 Key Generation

The keys are generated by choosing large integers $p, q \in \texttt{PRIME}$. We then calculate $N = p \cdot q$ and $\varphi(N) = (p-1) \cdot (q-1)$. Since 2015, the US National Institute of Standards and Technology recommends the minimum size for N to be 2048 bits. Hence, the magnitude of p, q can be assessed as 10^{600}. Next, a public exponent $e \in \mathbb{Z}^*_{\varphi(\mathbb{N})}$ is chosen so that e and $\varphi(N)$ are co-prime. The official recommendation for e is the prime number 65337. For this e we find the inverse in the cyclic field $\mathbb{Z}^*_{\varphi(N)}$ so that

$$d = e^{-1} \mod \varphi(N) \ .$$

To calculate d in polynomial time the extended euclidean algorithm can be used. The numbers d, e, N make up the keys to this encryption method. The public key is given in the format (N, e), while the private key is stored as (N, d). It is important for the private key to only be known to the entity, who is intended to be able to decrypt the messages. The public key needs to be known by anyone sending messages to the holder of the private key and should hence be stored in a public directory.

2.2 RSA Encryption

After generating the keys in this way, we can publish the public key (N, e). If this is used to encrypt a message, only someone who knows the corresponding private key can

reverse the computation in polynomial time, since this is the simple direction of a trap door function. Inverting the encryption function is assumed to be a hard problem, but can be made easy through the use of the private key [Sma16, page 168]. A message M can be encrypted into a ciphertext C using

$$C = M^e \mod N .$$

2.3 RSA Decryption

By default, decryption works in a similar way as encryption. Since d and e are modular inverses of each other in $\mathbb{Z}^*_{\varphi(N)}$, their multiplication with one another results in the neutral element of the field. Decryption works by applying

$$M = C^d \mod N .$$

When we look at this in detail the values can be denoted as

$$M = C^d \mod N = (M^e)^d \mod N = M^{ed} \mod N .$$

The Fermat–Euler theorem [Raj] states that if M and N are coprime, then

$$M^{\varphi(N)} = 1 \mod N .$$

Hence,

$$\varphi(N) = (p-1) \cdot (q-1)$$

and thus

$$e \cdot d = 1 \mod \varphi(N).$$

It follows that there is a k so that $e \cdot d = k \cdot \varphi(N)$. We can conclude

$$M^{ed} = M^{(ed-1)+1} = M \cdot M^{ed-1} = M \cdot M^{k\varphi(N)} = M \cdot 1^k = M \mod N .$$

This shows why d and e can be picked from $\mathbb{Z}^*_{\varphi(N)}$ and still be used to de- and encrypt information in modulo N [Sma16, pages 172–173].

2.4 The RSA Problem

The RSA problem describes finding the private key of an RSA encryption system knowing only the RSA encryption system and the public key in accordance with Kerckhoff's principle. To solve the problem an attacker needs to find a way to efficiently compute m given

$$C = M^e \mod N.$$

Considering the fact that N and e are public, we can assume that the attacker will use a chosen-plaintext attack since they can calculate the ciphertext to any message they want using the public key and the known encryption method. They also know that N is the product of two large prime numbers [Sma16, page 173].

2.4.1 RSA Not Harder Than Factoring

Assuming we can factor any N, we can compute p and q and thus $\varphi(N) = (p-1) \cdot (q-1)$ and following that use the extended euclidean algorithm to calculate

$$d = e^{-1} \mod \varphi(N).$$

After finding d we can decrypt M by calculating

$$C^d = M \mod N$$

from the definition of RSA. Therefore the RSA problem is at most as hard as the factorization problem [Sma16, page 172].

2.4.2 Difficulty Of The RSA Problem

Just as there is no evidence that integer factorization is hard to compute, there is no evidence that the RSA problem is any less hard. As we just showed, the RSA problem can not be harder than factorization, since we can solve it through factorization, but may well be simpler. An indicator that the RSA problem might be simpler than integer factorization is that the RSA problem only requires breaking one ciphertext, while integer factorization obtains the private key and thus breaks all possible ciphertexts. The search for d is equivalent in effort to the search for the prime factors of N, but knowing d is not a direct requirement for solving the RSA problem.

3 Key Recovery Algorithm

The Key Recovery Algorithm can recover the private key of an RSA cryptosystem after information about this key has been partially obtained. The key is stored redundantly in the RAM of the decrypting device. By recovering some of the bits of the key, the remaining bits can be calculated by the following algorithm using said redundancies.

3.1 Premises of a Key Recovery

In order for the key recovery algorithm to be used, partial information about the private key must be available. This partial information can be obtained, for example, by a cold boot attack on a machine to which there is physical access [HSH+09, page 49]. This information gives us knowledge about the total length of the individual bit strings as well as some bits, which we can then regard as known. Each memory device has a ground state. The bits we gain partial information about which are not in that ground state are considered confirmed in that state. All known bits have the same state. All bits that are not in this state must be assumed to be unknown, so both states can be considered correct. For efficient recovery of the full key, there must be a known δ-fraction of the entire key $(p, q, d, d_p, d_q, q_p^{-1})$. The density of known bits from the different values plays an essential role here. Thus the algorithm can be used with

$$\delta = 0.27 \text{ certainty over the bits of } p, q, d, d_p \text{ and } d_q$$
$$\delta = 0.42 \text{ certainty over the bits of } p, q, \text{ and } d$$
$$\delta = 0.57 \text{ certainty over the bits of } p \text{ and } q \text{ .}$$

Therefore the algorithm can be useful in combination with a cold boot attack [HS08, pages 1–2].

3.2 Cold Boot Attack

A cold boot attack is a side-channel attack in which an attacker with physical access to the target computer reads out the contents of the RAM after the system has been powered off. By cutting the power, software that may be running on the target device cannot delete sensitive data from RAM. Afterwards, an image of the RAM is created by imaging

tools[HSH+09, page 51]. It is based on the data remanence in common RAM modules, in which the bit states do not dissipate rapidly, but rather slowly. Depending on the computer, such remnants can be found after several seconds to minutes without power. The quality of the extracted data depends on the time it took to be extracted after the system has been powered off. Cooling the memory modules drastically extends the remanence time. Using liquid nitrogen to cool the modules the decay can be decelerated to only 0.00004% after ten minutes[HSH+09, page 49]. The cryptographic keys to encrypted data that were accessible at the moment of the system crash can then be extracted from the read data. The colder the storage is, the slower the decay. Thus the temperature and the time until the RAM module is powered up again affect the decay of the bits and thus the effectiveness of the attack. The dissipation of the stored information is not arbitrary, but each memory module has a ground state to which the bits return. Therefore, we can assume that every bit that is not in the ground state is confirmed, while those bits that are in the ground state must be considered unknown. This way we can partially recover the data and consider all bits not in ground state as confirmed [HS09].

3.3 Private Key Recovery

To recover the key the redundancy in the stored values $(p, q, d, d_p, d_q, q_p^{-1})$ can be used. The relevant formulas are

$$\begin{aligned} e \cdot d &= 1 \mod \varphi(N) \\ e \cdot d_p &= 1 \mod \varphi(p) \\ e \cdot d_q &= 1 \mod \varphi(q) \\ q \cdot q_p^{-1} &= 1 \mod p\,. \end{aligned}$$

Those can be transformed into these

$$e \cdot d = k\varphi(N) + 1 \tag{3.1}$$
$$e \cdot d_p = k_p\varphi(p) + 1 \tag{3.2}$$
$$e \cdot d_q = k_q\varphi(q) + 1 \tag{3.3}$$
$$q \cdot q_p^{-1} = k'p + 1 \tag{3.4}$$

for some $k, k_p, k_q, k' \in \mathbb{Z}$. The variations of k show how many times the order of the group has to be added to 1 until the original number before the modulo operation is reached. To solve the equations above the values k, k_p, k_q, k' need to be found. The fact that $d \in \mathbb{Z}^*_{\varphi(N)}$ obviously ensures $d < \varphi(N)$. It follows that $k < e$, since $k \geq e$ implies $e \cdot d <$

$k\varphi(N) + 1$ which is contradictive to $e \cdot d = k\varphi(N) + 1$. As $e = 65537$ in almost all RSA implementations, it is possible to try all candidates for k.
For each possible choice for $\hat{k}$, through the function

$$\tilde{d}(\hat{k}) = \lfloor \frac{\hat{k}(N+1)+1}{e} \rfloor$$

d can be approximated if $\hat{k}$ equals k. This process enables us to determine k and the more significant half of the bits of $\tilde{d}$. Once k is known k_p, k_q can be calculated through the congruences of the equations above.
Knowing k_p and k_q we can run the following algorithm to find p, q. Let $p[i]$ denote the ith bit of p starting from the least significant bit. Let $\tau(x)$ denote the exponent of the largest power of 2 that divides x. Since p, q are prime, they are uneven so $p[0] = q[0] = 1$. It follows that $2|p-1$, so $2^{1+\tau(k_p)}|k_p(p-1)$. Considering that $\varphi(p) = p-1$ we can reduce 3.3 $\mod 2^{1+\tau(k_p)}$ to

$$ed_p \equiv 1(\mod 2^{1+\tau(k_p)}) .$$

Knowing e we can immediately correct the $1 + \tau(k_p)$ least significant bits of d_p. Congenially we can proceed with 3.2 and 3.4 to correct the $2 + \tau(k)$ and $1 + \tau(k_q)$ bits of d and d_q. Furthermore, all the bits $< i$ are confirmed for p, which means that inaccuracies in more significant bits lead to changes in $d_p[i + \tau(k_p)]$ and not in p. The same goes for $q[i]$ affecting $d_q[i + \tau(k_q)]$. Either one causes changes in $d[i + \tau(k)]$.
After recovering the least significant bits for each of our five variables, we can try to recover the remaining bits. For each bit index i, we consider a slice of bits:

$$p[i] \qquad q[i] \qquad d[i + \tau(k)] \qquad d_p[i + \tau(k_p)] \qquad d_q[i + \tau(k_q)] .$$

To find the correct solution for slice i, all combinations that agree with that solution at all but the ith position needs to be computed. That way all possible combinations up to bit slice i will be enumerated. As already explained the only possible solution for slice $i = 0$ is already known, so we can start the algorithm. The solution for N will be found by the time $i = \lfloor \frac{N}{2} \rfloor$ has been reached.
Using the partial information, which has been gained through an attack, we can fill in (p', q', d', d'_p, d'_q) for each slice $i - 1$ to find all possible slices i. For that we repeatedly find

solving combinations for the equation system

$$\begin{aligned} p[i] + q[i] &\equiv c_1 \mod 2 \\ d[i + \tau(k)] + p[i] + q[i] &\equiv c_2 \mod 2 \\ d_p[i + \tau(k_p)] + p[i] &\equiv c_3 \mod 2 \\ d_q[i + \tau(k_q)] + q[i] &\equiv c_4 \mod 2 \,. \end{aligned}$$

Having five variables in an equation system with four equations leaves us up to two solutions for the respective ith bits of the variables. If we get two solutions for the ith bit, we need to calculate both possibilities for the $i+1$th bit. Using the information we gained about the private key, we can prune potential solutions that fulfill the equation system but contradict the gained information. Eventually a wrong branch is not going to have a valid allocation for the next bits and can be fully discarded as false [HS08].

3.4 Prospects

If we consider the system of equations over the private key values, we see that q_p^{-1} is not included in these systems. Therefore, q_p^{-1} is not considered for the preconditions given with the δ-fraction. To improve the algorithm, we wanted to find a way to consider the bits that can be obtained over q_p^{-1} in the algorithm as well. Our first goal was to understand q_p^{-1} so that we could extend the key recovery algorithm in such a way that the information obtained with q_p^{-1} is no longer discarded. In the algorithm used by Heninger and Shacham, no bit of information from q_p^{-1} is currently used to recover the private key. The hope was to find a way to include q_p^{-1} in the system of equations so that the information could be used directly in finding the result. This would have made the method faster, more efficient, and probably still usable even with a smaller number of known bits. The expected way for using this additional information was to use them to find k'. By finding k' we would have been able to recover the private key. While trying to find a way to accomplish this we found that given e and N, finding k' with $q \cdot q_p^-1 = k' \cdot p + 1$ is at least as hard as integer factorization.

4 Incorporating q_p^{-1}

In this chapter, we consider q_p^{-1} and its uses. The main goal is to understand how the value is calculated and what it is used for. We look at the dependencies to the other values in the private key. Then we work out how q_p^{-1} is used in the decryption and why it is used for it. Afterwards, we contemplate the possibilities q_p^{-1} offers to get information about other parts of the private key.

4.1 Structure Of q_p^{-1} With Regard To p And q

The private key is stored in the form $(p, q, d, d_p, d_q, q_p^{-1})$ to decrypt the ciphertexts. The value we focus on in this work is

$$q_p^{-1} = q^{-1} \mod p \,.$$

Hence,

$$q_p^{-1} \cdot q = 1 \mod p \,.$$

From this we can deduce that

$$q \cdot q_p^{-1} = k'p + 1$$

with k' being an unknown variable describing how many times the mod p needs to overflow until $q_p^{-1} \cdot q = 1$ is fulfilled. Since q_p^{-1} is calculated from other values with which it is being stored we can assume that the information is stored redundantly. This redundant way of storage is necessary for decrypting ciphertexts more efficiently. In the decryption algorithm, q_p^{-1} is used along with two other values in the Chinese Remainder Theorem to calculate the plain text from the ciphertext. Currently, it is not known, how k' can be reconstructed. Hence, the equation $q \cdot q_p^{-1} = k'p + 1$ can not be used. Our goal in this section is to understand the use of q_p^{-1} and its dependencies in the equation system of the key recovery algorithm. We investigate the behavior of the individual bits of number pairs $a, a^{-1} \in \mathbb{Z}_n^*$. We want to find peculiarities and possible dependencies of the bits with each other.

4.2 Chinese Remainder Theorem

The Chinese Remainder Theorem finds the linear congruency x for given pairwise coprime $v_i \bmod m_i$. Formally we are looking for an x with

$$x \equiv v_i \mod m_i \qquad \text{for } i = 1, \ldots, n \ .$$

4.2.1 RSA Decryption

In textbook RSA the decryption is done by exponentiating the encrypted message with the private part of the key d in the cyclic group of $\mathbb{Z}^*_{\varphi(N)}$, i. e.

$$M = C^d \mod N \ .$$

There is a more efficient way of decrypting, which is used in almost all practical applications of RSA due to being faster, more efficient, and hence cheaper. The RSA private key is stored in the format $(p, q, d, d_p, d_q, q_p^{-1})$ with the following properties:

$$\begin{aligned} p, q &\text{ are prime.} \\ N &= p \cdot q \\ 1 < e &< \varphi(N) \\ GCD(e, \varphi(N)) &= 1 \\ d &= e^{-1} \mod \varphi(N) \\ d_p &= d \mod (p-1) \\ d_q &= d \mod (q-1) \\ q_p^{-1} &= q^{-1} \mod p \end{aligned}$$

The public key is composed of (N, e), therefore those values are common knowledge. Using these properties of the different values stored as the private key the common way of decryption is to calculate

$$(c \mod p)^{d_p} \mod p \tag{4.1}$$

$$(c \mod q)^{d_q} \mod q \tag{4.2}$$

and use the Chinese Remainder Theorem on these two results and q_p^{-1}. This calculation is about four times faster than the naive calculation using exponentiation with a large number[HS08, page 3–6].

4.2.2 Garner's Algorithm

Using Garner's Algorithm, the Chinese Remainder Theorem can be simplified and solved faster. It is an efficient method to determine the linear congruency $x, 0 \leq x < M$ of given modular residues $v(x) = (v_1, v_2, \ldots, v_t)$. From the Chinese Remainder Theorem follows that residues of x are pairwise co-prime moduli $m_1, m_2, \ldots, m_t$, with $M = \prod_{i=1}^{t} m_i$. We compute the linear congruency so that $x \equiv v_i \bmod m_i$ for $i = 1, \ldots, t$.

Algorithm 1: Garner's Algorithm

```
for i from 2 to t do
    C_i = 1
    for j from 1 to i − 1 do
        u = m_j^{-1} mod m_i
        C_i = u · C_i mod m_i
u = v_1
x = u
for i from 2 to t do
    u = (v_i − x)C_i mod m_i
    x = x + u · ∏_{j=1}^{i−1} m_j
return x
```

This algorithm is an efficient form of the Chinese Remainder Theorem. It can be used as an alternative to the latter [MvOV01, page 612].

The Chinese Remainder Theorem formula for RSA decryption can be generalized using Algorithm 1. By inserting our given variables into the algorithm we can establish

$$\begin{aligned} c_2 &= m_1^{-1} \mod m_2 \\ c_3 &= m_1^{-1} \cdot m_2^{-1} \mod m_3 \\ c_4 &= m_1^{-1} \cdot m_2^{-1} \cdot m_3^{-1} \mod m_4 \\ C_i &= \prod_{u=1}^{i-1} m_j^{-1} \mod m_i \end{aligned}$$

for the first loop. After that we set

$$\begin{aligned} u &= v_1 \\ x &= u. \end{aligned}$$

Finally the last loop results in

$$u = (v_2 - v_1)C_2 \bmod m_2$$
$$x = v_1 + ((v_2 - v_1)C_2 \bmod m_2) \cdot m_1.$$

In the RSA context these equations are more commonly represented as

$$h = q^{-1}(m_1 - m_2)(\mathrm{mod} p)$$
$$m = m_2 + hq(\mathrm{mod} p \cdot q)$$

with

$$m_1 = c^{d_p} \mod p$$
$$m_2 = c^{d_q} \mod q.$$

It is important to notice, that the variable names have changed. Before the moduli were named m_i, but those labels were replaced by p and q and M with N, while m_i now replaces the v_i representation of the integer values of the Chinese Remainder Theorem. Using these equations RSA can be decrypted without using a large exponent but using two smaller exponents instead. For the large exponent, we can set $\varphi((p-1) \cdot (q-1))$ as the upper bound. For the small exponents, the upper bound is p or q respectively. Thus the bit length of the large exponent is approximately twice as large as that of the small exponents. This results in the increased efficiency of the decryption.

4.3 Bitwise Modular Inverse

The multiplicative inverse of a modulo b exists if and only if a and b are co-prime, that is, if $GCD(a, b) = 1$. If the multiplicative inverse of a number $a \bmod b$ exists, then one can define the operation of dividing any other number by $a \bmod b$, by multiplying that number by the inverse a^{-1}. If b is a prime number, then all numbers except zero are invertible, which makes the ring of integers modulo b a field. The values stored in the private key are $(p, q, d, d_p, d_q, q_p^{-1})$. Knowing the public key and any of these six values, all other values can be computed. Except for q_p^{-1}, the dependencies between the values are described by Nadia Heninger and Hovav Shacham[HS08]. The value q_p^{-1} is different from the other values in that it is modularly inverted. So if we get information about q_p^{-1} we must be able to verify the value bit by bit or use its bits as verification for the other partially known values in order to draw conclusions about the other elements of the stored private key.

4.3.1 Dependencies

Since the information about the private key is available bitwise, we try to find a way to find bitwise dependencies of a number and its inverse in their cyclic group. To implement this, for some $\mathbb{Z}_n^*$ we consider all elements of the group with their respective inverse and try to find statistical salience suggesting possible dependencies. We do this exemplarily for $\mathbb{Z}_{11}^*$.

Table 4.1: Inverses in $\mathbb{Z}_{11}^*$

a	a_2	a^{-1}	a_2^{-1}	Hamming Distance	Length Difference
1	1	1	1	0	0
2	10	6	110	1	1
3	11	4	100	3	1
4	100	3	11	3	1
5	101	9	1001	2	1
6	110	2	10	1	1
7	111	8	1000	4	1
8	1000	7	111	4	1
9	1001	5	101	2	1
10	1010	10	1010	0	0

From 4.1 we can see that there is no obvious regularity for any a and a^{-1}. Just looking at the least significant bits of the non self inverse values the distribution is seemingly random. The average Hamming distance is 2 and the average difference in length is 0.8. The Hamming distance is a measure of the difference of strings. It is defined as the number of bits that differ between the two compared strings. As there are two combinations of allocations for each bit that are identical and two differing allocations the expected outcome for a random distribution is $\frac{1}{2}$. Therefore the expected Hamming distance is half the bit length of n. With the Hamming distance being close to the random distribution, we can call it unobtrusive and negligible for further consideration. The average length difference between a and a^{-1} describes the difference of the positions of the leading 1 bit of each string. We will now compare the values of those results for different $\mathbb{Z}_n^*$.

Table 4.2: Distributions for different $\mathbb{Z}_n^*$

	$\mathbb{Z}_{11}^*$	$\mathbb{Z}_{17}^*$	$\mathbb{Z}_{56081}^*$	$\mathbb{Z}_{65537}^*$	$\mathbb{Z}_{86609}^*$
Bitlength	4	5	16	17	17
Average Hamming Distance	2	1.875	7.966	8	8.331
Average Length Difference	0.8	0.5	1.427	1.331	1.493
Same LSB Distribution	0.556	0.375	0.501	0.494	0.494

The selection of primes tested for 4.2 was arbitrary. The first two have been chosen due

to their small size. The $\mathbb{Z}^*_{65537}$ was chosen due to its use as e and it possibly being a special case because $65537 = 2^{16} + 1$. The other two numbers were chosen randomly to see if the oddity found regarding the similar lengths of numbers and their modular inverse were solely present in $\mathbb{Z}^*_{65537}$ or also in other numbers of greater lengths. The average Hamming distance shows the average over all Hamming distances of any $a \in n$ and $a^{-1} \bmod n$. The average length difference shows the average over the number of bits after and including the leading 1 in any $a \in n$ and $a^{-1} \bmod n$. The same LSB distribution shows the distribution of $a \in n$ and $a^{-1} \bmod n$ with the same least significant bit. From 4.2 it can be observed that the average Hamming distance and the occurence of the same least significant bits are correlated with the expected random distribution with very little deviation. At the same time, the average length difference between a number and its modular inverse is deviating significantly from the expected random distribution. Even with larger modular groups, it can be observed that a number and its inverse exceptionally often have the same magnitude. In the case of $\mathbb{Z}^*_{56081}$, the bit length is identical for more than $1/4$ of the numbers and differs by only one digit for another $1/3$. Thus, it might be possible to draw conclusions about the magnitude of q by the most significant bit of q_p^{-1}. However, if we think more carefully about the length distributions of the numbers in the groups, this striking distribution can be explained. Since we consider only $n \in \texttt{PRIME}$, all numbers $0 < a \leq n \in \mathbb{Z}^*_n$. This obviously applies to $a^{-1} \bmod n$, since it must also be an element of the group. Thus, half of *all* considered bit strings have the same length. Another quarter of all considered bit strings differs in length by one. This continues in such a way. This explains the striking frequency of numbers that are as long as their inverse.

4.3.2 Gained Information

Assuming we had the complete q_p^{-1} in the best-case scenario, we could use the probability distributions to predict the first bit of q. However, the information gain is minimal, since we can only predict how likely it is that the most significant bit is in a given position. In practice, however, this knowledge is irrelevant, since to guarantee a certain key length for p and q, the leading bit at the required position is set to 1. So any information that might be gained from these dependencies is already known.

4.3.3 Discussion And Open Problems

The goal of these considerations was to find a possibility to draw possible conclusions about p or q by partial bitwise knowledge q_p^{-1}. On the whole, the distribution of the bits of numbers and their modular inverses seems to be very random. We have looked more closely at the first and last bits and, in general, at the Hamming distance. Here mostly

the expected values were reached. Only the most significant bit or the length of the bit string deviated from these. However, these are the only abnormalities that would suggest any form of dependencies. The information gain on this way is even under optimal circumstances very small and at the same time redundant. A bitwise confirmation of q_p^{-1} by including the value in the key recovery algorithm via finding k', therefore, does not seem feasible.

4.4 Decrypting Modified Ciphertext

We use the method enabled by Algorithm 1 to decrypt the ciphertext instead of textbook exponentiation, which uses d_p, d_p, and q_p^{-1}. By substituting our known values and ciphertext C in

$$\begin{aligned} m_1 &= C^{d_p} \mod p \\ m_2 &= C^{d_q} \mod q \\ h &= (q_p^{-1} * (m_1 - m_2)) \mod p \\ m &= m_2 + h * q \end{aligned}$$

it gets decrypted into the message M. These formulas are derived from Algorithm 1. Our goal is to consider the behavior of the decrypted message $\tilde{M}$ when, given arbitrarily chosen pairs (M, C), we modify the ciphertext. An attacker on RSA has the ciphertext available for any plaintext he chooses since he can generate it himself with the public key. The private key is unknown to him, so we assume that the attacker cannot make any changes within the decryption. Thus, the attacker is only able to choose (M, C) arbitrarily. Hence, we do not change C during the decryption process but only before it. To observe the behavior of the decryption with modified ciphertexts, we choose a key pair arbitrarily, so we can try to attack the private key. This gives us the advantage over a chosen-plaintext attack to be able to also have any $\tilde{C}$ decrypted.

4.4.1 C+1

We consider the change in plaintext with $\tilde{C} = C + 1$. We randomly choose

$$p = 359 \qquad q = 223 \qquad e = 17$$

and generate from it

$$d = 74801 \qquad d_p = 337 \qquad d_q = 209 \qquad q_p^{-1} = 293\ .$$

This results in $N = p \cdot q$ as public key $(80057, 17)$ and as private key $(80057, 74801)$. In the redundant representation used, the private key is stored as $(359, 223, 74801, 337, 209, 293)$. Randomly choosing $(M = 9247, C = 31088)$ we get $\tilde{C} = 31089$ and $\tilde{M} = 64986$. Comparing M and $\tilde{M}$, we notice that both numbers are very different. They are far from each other and there is no visible similarity. Even neglecting the modulo, C^d and $\tilde{C}^d$ are very different. The number $\tilde{C}^d$ is one decimal place longer. The numbers do not divide each other by an integer. Likewise, they do not divide an integer by values of the key. Next, we put elements of the key as a message, encode them, and look at the change in $\tilde{M}$ compared to M.

Table 4.3: Modified C with M as parts of the key

	M	$\tilde{M}$	C	$\tilde{C}$
p	359	9335	23694	23695
q	223	27207	61325	61326
d	74801	54111	46979	46980
e	17	63508	64284	64285
d_p	337	51023	59498	59499
d_q	209	38318	36285	36286
q_p^{-1}	293	27699	70852	70853

Looking at the results in table 4.3, we see that there are no similarities between the respective decrypted M and $\tilde{M}$. Since we can not find any useful results already for a small number space, it makes no sense to consider further values in this range.

4.4.2 2C

With the same keys and the same M we now consider $\tilde{C} = 2C \mod N$. Thus, in modular space, for $M = 9247$ we obtain the modified value $\tilde{M} = 22045$. Neglecting the modulo, we observe that

$$C^d = \tilde{C}^d / 2^d \, .$$

However, this dependence is easy to understand because of the way exponentiation works and the nature of $\tilde{C}$. Aside from this, there is no information to be gained from using $\tilde{C} = 2C$.

4.4.3 Conclusion

Changing the ciphertext is a common attack. Success here might have benefited us more in solving the RSA problem than in improving the key recovery algorithm. There is no

significant result to discuss. We only found that even minimal changes to the ciphertext cause the decryption to produces a heavily modified message that reveals no information about the original message. This is obvious since changes to the base at a power as high as d can cause the results to reach different magnitudes.

5 Factorizing N From k'

One way to solve the RSA problem is to find the private key (N, d). Since N is known, the problem is to find d. This value is the modular inverse of the known public exponent e in $\mathbb{Z}^*_{\varphi(N)}$. So to calculate it we need the factorization of N so that we can find $\varphi(N)$. Factorization of integers is believed to be a computationally complex problem. It is currently unknown whether an efficient non-quantum algorithm for factoring integers exists. However, it is also not proven that there is no solution to this problem in polynomial time.

5.1 Coppersmith Method

The Coppersmith Method is a theorem that efficiently finds all roots of normalized polynomials of a certain modulo. This method is effective if the public exponent is small enough, or when information on the private key is available.

5.1.1 Purpose

Theorem 5.1 (Coppersmith). *Let $f \in \mathbb{Z}[x]$ be a monic polynomial of degree d and N an integer. If there is some root x_0 of f modulo N such that $|x_0| \leq X = N^{1/d-\varepsilon}$ then one can find x_0 in time a polynomial in log_N and $1/\varepsilon$, for fixed values of d [Sma16, page 281].*

The algorithm's purpose is to find an x fulfilling a monic polynomial of the form

$$f(x) = f_0 + f_1 x + \cdots + f_{d-1} x^{d-1} + x^d$$

over integers of degree d, while knowing there is an integer root $x_0 \bmod N$.

5.1.2 Fundamentals

The following is inspired [Sma16, pages 279-281]. For this purpose, we assume that there exists an $x_0 \bmod N$ for which it holds that $|x_0| < N^{1/d}$. With the help of the method, it is possible to find an x_0 efficiently. The basic idea is to find a polynomial $h(x) \in \mathbb{Z}[x]$ that has the same root in modulo N as the polynomial $f(x)$. This $h(x)$ should be small in the

sense that the norm of its coefficients

$$\|h\|_2^2 = \sum_{i=0}^{deg(h)} h_i^2 \tag{5.1}$$

is small. By finding an $h(x)$ fulfilling this we can simplify the problem of finding a root in the modular field N to finding a root over the integers. Let $h(x) \in \mathbb{Z}[x]$ denote a polynomial of degree $\leq n$ and let X and N be positive integers.

Lemma 5.2. *Suppose*

$$\|h(xX)\|_2 < N/\sqrt{n} \tag{5.2}$$

then if $|x_0| < X$ *satisfies*

$$h(x_0) = 0 \mod N \tag{5.3}$$

then $h(x_0) = 0$ *over the integers and not just modulo N.*

Thus we have simplified the hard problem of finding a root in the modular field N to finding the root for the polynomial, for which there is an efficient solution. The Manhattan norm is also referred to as the 1 norm. The distance derived from this norm is called the Manhattan distance or the 1-norm distance. The 1-norm is simply the sum of the absolute values of the vector. We note it as

$$\|z\|_1 = \sum_{i=1}^{m} |z_i| \ .$$

The Euclidean norm or 2-norm is the root of the sum of the squares of all values of the vector. We note it as

$$\|z\|_2 = \sqrt{\sum_{i=1}^{m} z_i^2} \ .$$

Proof (for lemma 5.2). From 5.2 and the condition $|x_0| < X$ with 5.3 we can deduce that

$$h(x_0) \overset{5.1}{=} \sum_{i=1}^{m} (h_i x_0^i) \leq \sum_{i=1}^{m} |h_i x_0^i| \overset{x_0 < x}{<} \sum_{i=1}^{m} |h_i x^i| \overset{5.1}{=} \|h(xX)\|_1 \ .$$

Using the Cauchy-Schwarz inequality we can show that

$$\|h(z)\|_1 = \sum_{i=1}^{m}(|z_i| \cdot 1) \leq \sum_{i=1}^{m} z_i^2 \sum_{i=1}^{m} 1^2 = \sqrt{n} \cdot \|z\|_2 \ .$$

Hence,

$$\|h(xX)\|_1 \leq \sqrt{n} \cdot \|h(xX)\|_2 \leq \sqrt{n} \cdot N/\sqrt{n} = N \ . \qquad \square$$

This shows that if $h(x_0)$ is a root in the space of natural numbers and $x_0 < N$, then $f(x_0)$ is also a root in the group $\mathbb{Z}_N^*$. Returning to the polynomial $f(x)$ of degree d we notice

$$f(x_0) = 0 \mod N \tag{5.4}$$

also implies

$$f(x_0)^k = 0 \mod N \ . \tag{5.5}$$

For some given values m, u, v with $0 \leq u < d$ and $0 \leq v \leq m$

$$g_{u,v}(x) = N^{m-v} x^u f(x)^v \tag{5.6}$$

so

$$g_{u,v}(x_0) = 0 \mod N^m \ .$$

With a chosen m we try to find $a_{u,v} \in \mathbb{Z}$ so that

$$h(x) = \sum_{u \geq 0} \sum_{v=0}^{m} a_{u,v} g_{u,v}(x)$$

fulfills the condition 5.1. So we try to find integer values for $a_{u,v}$ so that the resulting polynomial h satisfies

$$\|h(x)\|_2 \leq N^m/\sqrt{d(m+1)}$$

with

$$h(xX) = \sum_{u \geq 0} \sum_{v=0}^{m} a_{u,v} g_{u,v}(xX) \ .$$

By doing this we remove the complexity of finding a root in modulo while maintaining equity in the least polynomials.

5.1.3 Forming The Polynomial $f(x)$

To utilize Coppersmith Method for us, we first need to rearrange the equations related to k' into a monic polynomial that the algorithm can be applied to. We know that

$$q_p^{-1} \cdot q = k' \cdot p + 1$$

applies for some k'. If we multiply this by p we get

$$q_p^{-1} \cdot N = k' \cdot p^2 + p \ .$$

Therefore, we look at

$$f(x) = k' \cdot x^2 + x \tag{5.7}$$

while knowing

$$f(p) = 0 \mod N \ .$$

On this equation, the Coppersmiths method can be applied to reveal p and through that also q. With k' we constitute the number of overflows over the upper bound in $\mathbb{Z}_p^*$ happen until $q_p^{-1} \cdot q$ reach the neutral element of the field. Next, we form the monic polynomial

$$f(x) = x^2 + ax + b$$

by reshaping 5.7 by dividing it by $k' \bmod N$ and hence assigning $b = 0$ and $a = (k'^{-1} \bmod N)$. From this, we can define the function $f(x)$ for our specific case with

$$f(x) = x^2 + k'^{-1} \cdot x + 0 \ .$$

Now, the Coppersmith Method is capable of finding a h with the same small roots over the integers as f has over the modular field.

5.1.4 Finding The Polynomial $h(x)$

To apply Coppersmith Method in the univariate case, we can choose

$$m = \lceil \frac{1}{d \cdot \varepsilon} \rceil$$

and $\varepsilon = \frac{1}{2} - log_N(p)$ [May10, page 12].
In this case, we chose $m = 2$ for 5.6 and compute

$$\begin{aligned}
g_{0,0}(xX) &= N^2 \\
g_{1,0}(xX) &= XN^2x \\
g_{0,1}(xX) &= bN + aXNx + NX^2x^2 \\
g_{1,1}(xX) &= bNXx + aNX^2x^2 + NX^3x3 \\
g_{0,2}(xX) &= b^2 + 2baXx + (a^2 + 2b)X^2x^2 + 2aX^3x^3 + X^4x^4 \\
g_{1,2}(xX) &= b^2Xx + 2baX^2x^2 + (a^2 + 2b)X^3x^3 + 2aX^4x^4 + X^5x^5 \ .
\end{aligned}$$

We are looking for a linear combination of the polynomials above so that the resulting polynomial has small coefficients as in 5.1. For that, we look for small vectors in the lattice generated by the columns of the following matrix A. Each column represents one of the polynomials above and each row represents a power of x. So the resulting matrix is

$$A = \begin{pmatrix}
N^2 & 0 & bN & 0 & b^2 & 0 \\
0 & XN^2 & aXN & bNX & 2abX & Xb^2 \\
0 & 0 & NX^2 & aNX^2 & (a^2 + 2b)X^2 & 2abX^2 \\
0 & 0 & 0 & NX^3 & 2aX^3 & (a^2 + 2b)X^3 \\
0 & 0 & 0 & 0 & X^4 & 2aX^4 \\
0 & 0 & 0 & 0 & 0 & X^5
\end{pmatrix} .$$

This matrix' determinant $det(A)$ is N^6X^{15} [Sma16, page 293]. By applying the Lenstra-Lenstra-Lovasz algorithm [LLL82] on this matrix we obtain the basis of lattice B with its first vector b_1 satisfying

$$\|b_1\| \leq 2^{6/4} det(A)^{1/6} = 2^{3/2} N X^{5/2} \ .$$

Setting $b_1 = Au \Leftrightarrow u = A^{-1}b_1$ with $u = (u_1, u_2, \ldots, u_6)^t$ we form the polynomial

$$h(x) = u_1 g_{0,0}(x) + u_2 g_{1,0}(x) + \cdots + u_6 g_{1,2}(x)$$

so,

$$\|h(xX)\| \leq 2^{3/2} N X^{5/2} .$$

To be able to ensure that we search for a root and not a modular root we require that

$$2^{3/2} N X^{5/2} < N^2/\sqrt{6} .$$

By following this we determine an integer root of $h(x)$, which at the same time is the small root x_0 of $f(x)$ modulo N under the premise that

$$|x_0| \leq X = \frac{N^{2/5}}{48^{1/5}} . \tag{5.8}$$

With these particular parameters the algorithm will work if $x_0 \leq N^{0.39}$. To solve a more arbitrary version of this problem a bigger m needs to be chosen when creating the $g_{u,v}$ functions and matrix A [Sma16, pages 279–282].

5.1.5 Finding The Integer Root

There are multiple algorithms for finding the roots of a polynomial. The number or roots is limited by the degree of the polynomial. For example the Durand-Kerner method can be used to find all real roots of h [Kho18].

5.2 Examplary Application

We now demonstrate this procedure using an example. The goal is to show that the Coppersmith Method finds a solution. First, we generate random values for this purpose. To keep the numbers manageable, we use a 32-bit integer prime number for p instead of large prime numbers that would be suitable for RSA. The integer q is a prime number at least 150000 times larger than p. We add this specification for p and q to require less precision in finding the polynomial h and for ease of presentation. As stated above chosing $m = 2$ will find a result for $x_0 \leq N^{0.39}$, so we need to chose our values according to this condition. The parameters used in the example are

$$\begin{aligned} p &= 1044502237 & q &= 156675335550007 \\ N &= 163647738464707936865659 & e &= 65537 \\ d &= 110346173971500412395161 & d_p &= 634396013 \\ d_q &= 81623601655307 & q_p^{-1} &= 447643816 . \end{aligned}$$

Suppose that $k' = 67146572400003$ is known. Similarly, the public key (N, e) is known. Since we are trying to find a polynomial $f(x) = x^2 + ax + b$ for the next step, we can now see the correct shape. We assume that

$$\begin{aligned} f(x) &= k' \cdot x^2 + x \qquad | : k' \mod N \\ \overset{f(x_0)=0}{\Longrightarrow} f(x) &= x^2 + a \cdot x \end{aligned}$$

with

$$\begin{aligned} a \cdot k' &= 1 \mod N \\ a &= k'^{-1} \mod N \\ a &= 67146572400003^{-1} \mod 163647738464707936865659 \\ a &= 32858047726877786792068 \, . \end{aligned}$$

From 5.8 we can compute that $|x_0| \leq X = \frac{N^{2/5}}{48^{1/5}} \approx 889847850$. Furthermore, $b = 0$. After we know all these variables, we can determine the elements in A. On this matrix A we can apply the Lenstra-Lenstra-Lovasz algorithm to obtain B. The vector

$$b_1 = \begin{pmatrix} 0 \\ -6942456015458774152552641864334257046213215 0 \\ -6037826200338042912771962773661340270053100 00 \\ 6021155816738861182635724585298834354325000 00 \\ -6867109741598217147713749162515627013125000 00 \\ 5579287964713576612038160975718676740625000 00 \end{pmatrix}$$

is thus the first column of B. Knowing this and A we can solve

$$u = A^{-1} b_1$$

for vector u. Finally we compute

$$u = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{pmatrix} = \begin{pmatrix} 0 \\ 435258246905883016914021099444603550077863 45 \\ -216778027551515425174347933678851790149301012 \\ 19792231360190267028828 \\ -65716095453756668828346 \\ 1) \end{pmatrix} .$$

We create polynomials

$$\begin{aligned}
g_{0,0}(x) &= N^2 \\
g_{1,0}(x) &= N^2 x \\
g_{0,1}(x) &= bN + aNx + Nx^2 \\
g_{1,1}(x) &= bNx + aNx^2 + Nx^3 \\
g_{0,2}(x) &= b^2 + 2bax + (a^2 + 2b)x^2 + 2ax^3 + x^4 \\
g_{1,2}(x) &= b^2 x + 2bax^2 + (a^2 + 2b)x^3 + 2ax^4 + x^5 \ .
\end{aligned}$$

Finally, we calculate the polynomial

$$h(x) = g_{0,0}(x) \cdot u_1 + g_{1,0}(x) \cdot u_2 + g_{0,1}(x) \cdot u_3 + g_{1,1}(x) \cdot u_4 + g_{0,2}(x) \cdot u_5 + g_{1,2}(x) \cdot u_6 \ .$$

It is easy to find integer roots for polynomials. For $h(x_0) = 0$ we find $x_0 = 1044502237$ with

$$\begin{aligned}
f(x_0) &= 0 \mod N \\
f(1044502237) &= 0 \mod 163647738464707936865659 \ .
\end{aligned}$$

This root for $f(x_0)$ in modulo N is the p we are looking for. From this we can calculate $q = \frac{N}{p}$. To use this to compute the private key, we first compute

$$\varphi(N) = (p-1) \cdot (q-1) \ .$$

We can extract d from the known e by

$$d = e^{-1} \mod \varphi(N)$$

using the extended Euclidean algorithm. By doing so, we not only solved the RSA problem at hand, but also found the private exponent of the key, breaking the factorization for this system.

5.3 Summary

In this chapter, we demonstrate that it is possible to find the integer factorization of N if k' can be obtained somehow. We show this by assuming that k' is known. By simply rearranging the known equations, we are able to put the values into a form that we can calculate using Coppersmiths method. This means that we represent our searched p in a

certain way in a polynomial, so that by finding a root for this polynomial we can determine p. However, since we are in a cyclic group, we must first solve the problem of finding a root in this group, which in itself is not an easy problem. However, through Coppersmith we are able to create another polynomial h according to certain criteria, for which we only need to find a root in the space of integers. Solving this is easy. Because of the choice of h in relation to f, it holds afterwards that one of the integer roots of h is also a root of f and smaller N. Even if several values satisfy this criterion, their number is finite and small, so that the correct value can be found quickly. This root is the p that was used to create the keys. Once p is known, the computation of the other elements of the key is trivial.We have demonstrated this first in general, later for a specific example.

6 Conclusions

6.1 Summary

The original goal of this work was to improve the key recovery algorithm developed by Heninger and Shacham. The reasoning behind this was that for the recovery from the partially recovered values $(p, q, d, d_p, d_q, q_p^{-1})$, they only used the partial information about the first five values. Information about q_p^{-1} is also collected during the recovery of the partial information, but is not brought into the algorithm. We have thus seen an opportunity here to improve the algorithm in also using that information.

To create a broad basis for understanding, we first look at the underlying procedures that are relevant for this. These are the RSA encryption method on the one hand and the actual key recovery algorithm on the other.

For RSA, we look at key generation, how encryption and decryption work, and why this works. We then briefly look at the effort required to solve the RSA problem.

The key recovery algorithm is the heart of the work and is shown accordingly in detail. Basically, the algorithm works by the fact that p, q, d, d_p and d_q are also bitwise dependent on each other, so that starting with the least significant bit assignments for the five values can be found by a system of equations. Thus, the values can be calculated bit by bit. Because of the bitwise partial information about these values, wrong assignments can be pruned efficiently.

Afterwards we think about q_p^{-1}. We want to see how it is calculated, what it is used for in the decryption and why it is in memory at all. Furthermore we look at possibilities to find dependencies to the other values of the algorithm. Since the key recovery algorithm uses a system of equations that computes the values bit by bit, the first idea is to determine if this can work for q_p^{-1}. However, the problem is that this value is modularly inverted. By analyzing some modular groups, we find that no statement about the bit locations of the inverse to a number can be made based on the bits of that number, which is much better than guessing. Trying to modify the ciphertext before decryption and find possible pairs of message un ciphertext where something striking happens, we have no success.

During the considerations the equation

$$q_p^{-1} \cdot q = k' \cdot p + 1$$

caught our eye. This can be transformed by multiplying by p so that

$$q_p^{-1} \cdot N = k' \cdot p^2 + p$$

is obtained. The right-hand side of the equation can be transformed into a polynomial such that p can be recovered by Coppersmith method. We describe this procedure and also present it with an example.

6.2 Discussion and open problems

After several attempts to find a way to gain information about k', we have come to the conclusion that we cannot solve this task within the scope of this bachelor thesis. The dependencies of q_p^{-1} with the other values of the private key are not usable by the modular inversion with the information available to us. Because we know only for certain bits from the values whether they are correct, we cannot use the value of q_p^{-1} to include it like the other values in a bitwise acting system of equations. Therefore, we did not find a way to calculate k'. However, we did manage to show that it is possible to compute the private key if we succeed in finding k'. Once all of k' is known the Coppersmith method can be used to find p and through that the whole key. Since it is possible to solve the integer factorization for N in linear time using k' and thus also solve arbitrary RSA problems, we can assume that finding k' is at least as hard as the integer factorization.
Hence, the main problem remains. The bits of q_p^{-1} obtained by an attack are not used in the algorithm. If we know the values q_p^{-1} or k', we can fully recover the key from them. However, it is not enough to have partial information, because through it we are not able to infer the unknown parts of the bit string.

References

[HS08] Nadia Heninger and Hovav Shacham. Reconstructing RSA private keys from random key bits. *IACR Cryptol. ePrint Arch.*, 2008:510, 2008.

[HS09] Nadia Heninger and Hovav Shacham. Reconstructing RSA private keys from random key bits. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, volume 5677 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009.

[HSH+09] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, 2009.

[Kho18] Dmitry I. Khomovsky. Generalizations of the durand-kerner method. *CoRR*, abs/1806.06280, 2018.

[LLL82] A. K. Lenstra, H. W. Lenstra, and L. Lovasz. Factoring polynomials with rational coefficients. *MATH. ANN*, 261:515–534, 1982.

[May10] Alexander May. Using lll-reduction for solving RSA and factorization problems. In Phong Q. Nguyen and Brigitte Vallée, editors, *The LLL Algorithm - Survey and Applications*, Information Security and Cryptography, pages 315–348. Springer, 2010.

[MvOV01] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.

[Raj] Wissam Raji. *Theorems of Fermat, Euler, and Wilson*. https://math.libretexts.org/@go/page/8835.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[Sma16] Nigel P. Smart. *Cryptography Made Simple*. Information Security and Cryptography. Springer, 2016.

Sage Code: Coppersmith Method

Listing 1: Coppersmith

```
#Variable/ Key Generation
p=random_prime(2^32, lbound=2^16)
q=next_prime(150000*p)
N=p*q
e=65537
d=inverse_mod(e, (p-1)*(q-1))
dp=d%(p-1)
dq=d%(q-1)
qi=inverse_mod(q, p)
X=round(N^(2/5)/48^(1/5))
kqi=(qi*q-1)/p
a=(1/kqi)%N  #inverse_mod(kqi, N)
b=0

#Coppersmith
A = Matrix([[N^2,0,b*N,0,b^2,0],
[0,X*N^2,a*X*N,b*X*N,2*a*b*X,X*b^2],
[0,0,N*X^2,a*N*X^2,(a^2+2*b)*X^2,2*a*b*X^2],
[0,0,0,N*X^3,2*a*X^3,(a^2+2*b)*X^3],
[0,0,0,0,X^4,2*a*X^4],
[0,0,0,0,0,X^5]])
#We need to transpose A before using LLL because Sage handles basic vectors
#differently than our method does
B=A.transpose().LLL().transpose() # Transpose matrix twice to restore original form
b1=B.column(0)
#"b1␣in␣B␣will␣satisfy␣||b_1||\leq␣2^(3/2)*NX^(5/2)"

#Prepare variables for h(x)
u=A.inverse()*b1
g1(x)=N^2
g2(x)=N^2*x
g3(x)=b*N + N*x^2 + a*N*x
g4(x)=b*N*x + N*x^3 + a*N*x^2
g5(x)=b^2 + 2*a*b*x +  x^4 + 2*a*x^3 + (a^2+2*b)*x^2
g6(x)=b^2*x + 2*a*b*x^2 + x^5 + 2*a*x^4 + (a^2+2*b)*x^3

#Create h(x)
```

Sage Code: Coppersmith Method

```
h1(x) = u[0]*g1(x)
h2(x) = u[1]*g2(x)
h3(x) = u[2]*g3(x)
h4(x) = u[3]*g4(x)
h5(x) = u[4]*g5(x)
h6(x) = u[5]*g6(x)

h=h1+h2+h3+h4+h5+h6

#find integer roots of h
print(h.roots(ring=ZZ))
}
```