Menemui Matematik (Discovering Mathematics)
Vol. 45, No. 2: 177 - 184 (2023)

Uncover Weak Keys of RSADemo2 Software Using Batch Greatest Common Divisor Algorithm

Luqman Hakeem Zainul Hisham[1], Muhammad Asyraf Asbullah[2] and Saidu Isah Abubakar[3]

[1]*Department of Mathematics and Statistics, Universiti Putra Malaysia, 43400 UPM Serdang, Malaysia*
[2]*Institute for Mathematical Research, Universiti Putra Malaysia, 43400 UPM Serdang, Malaysia*
[3]*Department of Mathematics, Sokoto State University, Airport Road Sokoto, PMB 2134, Nigeria*
[1]200581@student.upm.edu.my, [2]ma_asyraf@upm.edu.my, [3]siabubakar82@gmail.com

ABSTRACT

RSA encryption is crucial for protecting our privacy and allowing us to transfer information over the internet without being concerned that unauthorised people will see it. RSA encryption can be deemed insecure in a variety of ways. This project focuses on factoring RSA moduli to detect weak RSA keys generated by an RSA simulator software called RSADemo2 (v2.1) (Darby, 2016). The approach used to factor RSA moduli is collecting samples from RSADemo2 and then using the batch-greatest common divisor (gcd) method. In addition, we will be using Maple software to simulate the batch-gcd algorithm on the collected samples.

Keywords: encryption, RSADemo2, greatest common divisor, cryptography weak keys

INTRODUCTION

Cryptography is a centuries-old science that involves writing secret codes. Some experts believe cryptography arose spontaneously following the invention of writing, with applications ranging from diplomatic messages to wartime battle plans. New types of cryptography emerged shortly following the widespread development of computer communications. Nowadays, cryptography is necessary when communicating over any untrustworthy medium, especially when discussing our privacy, such as our work and personal details like our name and address.

Public Key Cryptography (PKC) is the most important new cryptographic advancement discovered by Diffie and Hellman (1976). It is called asymmetric encryption because a two-key cryptosystem is used in which two people can communicate securely across an insecure channel without sharing the secret key. PKC relies on one-way functions, which are simple but difficult to compute for their inverse. The crucial point is that it does not matter which keys are used first, but both are needed for the process to work (Diffie and Hellman, 2022). RSA is a cryptographic algorithm named after the initials of three creators: Rivest, Shamir, and Adleman (Rivest et al., 1978). RSA was inspired by the earlier published works of Diffie and Hellman (1976), who described the idea of such an algorithm but never fully developed it. RSA implemented two crucial ideas: public key encryption and digital signature, and later, such cryptographic algorithm was made to replace the less secure National Bureau of Standards (NBS) algorithm (Milanov, 2009).

Heninger et al. (2012) conducted the largest network survey analyzing the Internet's public keys to detect RSA weak keys of Transport Layer Security (TLS) and Secure Shell (SSH) servers. They proved that weak keys are surprisingly widespread. Even more worryingly, these researchers extracted RSA private keys for 0.50% of TLS hosts, 0.03% of SSH hosts, and 1.03% of SSH hosts, thanks to nontrivial common factors between their public keys. The study's first part is the most thorough Internet-wide analysis of two of the most significant cryptographic protocols, TLS and SSH. They gathered 6.2 million distinct SSH host keys from 10.2 million hosts and 5.8 million different TLS certificates from 12.8 million hosts. At least 5.57% of TLS hosts and 9.60% of SSH hosts appear vulnerable because they share the same keys. The study was able to compute the private keys for 64,000 (0.50%) TLS hosts and 108,000 (1.06%) SSH hosts using only the scan data they collected by exploiting known flaws in the RSA cryptosystem. Finding the largest common divisor makes it simple to compute the private keys of distinct moduli that share exactly one prime factor in the case of RSA, although the public keys appear distinct.

Lenstra et al. (2012) performed a sanity check on the public keys they had gathered online. The main objective was to test the validity of the presumption that distinct random choices are made each time keys are created. Most public keys work as intended, but two out of every one thousand collected RSA moduli provide no security. In this paper, they contribute extra features in such a way as to improve their qualities from previous studies. These issues are usually taken for granted, focusing on actual public keys' randomness and computational characteristics. The fact that 12720 of the 4.7 million individual 1024-bit RSA moduli they initially gathered share a single large prime factor is very concerning. It is even worse because it is a trend that has been around for a while. From the current 11.4 million RSA moduli collection, 26965 are vulnerable, including ten 2048-bit ones. When used illegally, it may weaken the level of security that the public key infrastructure is designed to provide.

In 2003, Taiwan launched an e-government effort to offer all residents access to national public-key infrastructure. The cards create RSA keys, which are then digitally certified by a government agency and stored in a Citizen Digital Certificates database online. Unfortunately, some of these smart cards' random-number generators have produced legitimate certificates with keys that offer zero security because of serious design flaws. The method used to generate the secret keys for 184 different certificates is described in Bernstein et al. (2013). The simple gcd attack reveals that 103 keys factor into 119 primes on the whole Citizen Digital Certificates database. First, there is enough information in the shared primes to create a model of the prime generation procedure. Extrapolation from these prime factors is the next step: They propose a specific failure model in randomness generation compatible with 18 common divisors. The same model can produce 164 primes and verify each prime via batch trial division factors additional keys. Among the 184 keys are 103 with shared primes that can be effectively factored using a batch-gcd computation. The same kind of computation was used by Heninger and Shacham (2009) to factor tens of thousands of cryptographic keys on the Internet.

Our Contribution. Generating a lot of RSA keys can cause several keys to share the same prime factors whenever insufficient randomness happens. Factoring two RSA moduli that share the same prime factor can be very easy using the greatest common divisor (gcd), which means anyone with knowledge can factor those numbers, not just the owners. Therefore, data leak occurs because some servers or websites share different public keys but somehow share the same prime factor, which can be used to steal information or impersonate others, which are very dangerous. To detect weak RSA keys in a large set of moduli, we need to compute them using the gcd algorithm, and it will take much time and computing power depending on the size and length of the keys. In this work, we collect samples of RSA public keys generated via an RSA simulator application called RSADemo2 (v2.1) by Darby (2016). Then simulate the batch-gcd algorithm using Maple software on the collected samples.

METHODOLOGY

This section will discuss the batch's greatest common divisor method, introduced by Heninger et al. (2012), to factorize large moduli so that weak RSA keys can be detected. Firstly, let $N_1, N_2, ..., N_m$ be distinct RSA moduli. We want to find all moduli in this set that have a non-trivial factor in common with other moduli. Quasi-linear gcd computations are used in batch-gcd algorithms. This gcd computation is slow for small numbers but becomes more efficient as the numbers increase. According to Heninger et al. (2012), the following lemma is used:

Lemma 1 (Heninger et al. (2012)). Let $N_1, N_2, ..., N_m$ be distinct RSA moduli. Then $\gcd(N_1, N_2 \cdot ... \cdot N_m) = \gcd\left(N_1, \frac{N_1 \cdot N_2 \cdot ... \cdot N_m \left(\bmod N_1^2\right)}{N_1} \right).$

This approach aims to efficiently compute all gcd by doing a single massive multiplication. Before using this lemma, a product tree is used to compute the product of all moduli. The only way we can accomplish this is by computing all pairwise $\gcd(N_i, N_j)$ where $i \neq j.$ After that, we compute the product of all numbers,

$$N = N_1 \cdot N_2 \cdot ... \cdot N_m \tag{1}$$

Then, to see if it shares any factors with others, see whether,

$$\gcd(N_i^2, N) > N_i \tag{2}$$

First, we compute the product of all moduli, and after that, we will compute the gcd of (1) with all N_i^2 squared. We follow Kraft and Washington (2018) for most of the part related to mathematical correctness of computing gcd. Next, the batch-gcd is divided into three phases as follows.

Phase 1: Using product tree to compute product N efficiently.
The important point to remember is that while multiplication is associative, its execution time is not. It means that computing the product of

$$(N_1 \cdot N_2)(N_3 \cdot N_4)...(N_{m-1} \cdot N_m) \tag{3}$$

is faster than computing it from left to right, i.e.

$$\left((((N_1 \cdot N_2) \cdot N_3) \cdot N_4)... \cdot N_{m-1}) \cdot N_m\right) \tag{4}$$

In general, computing a product of n-bit numbers can be completed reasonably by multiplying balanced inputs (i.e., nearly equal sizes) whenever possible. It encourages the computation to be organized in a tree, which is commonly called the product tree.

Phase 2: Using a remainder tree to compute $N (\bmod N_i^2)$ efficiently.
After we compute the huge product of $N,$ we will then compute $\gcd(N_i^2, N)$ for all i's, however, if we do not perform it properly, it will be time-consuming. First, we observe that,

$$\gcd(N_i^2, N) = \gcd\left(N_i^2, N (\bmod N_i^2)\right) \tag{5}$$

Rather than computing m gcd's directly, where one number is huge, we conduct m modular reductions first, then m instances of $2n-$ bit gcd's. We will be using the following simple equation; $N(\bmod a) = \left(N(\bmod ab)\right)(\bmod a)$. Again, we will be using the product tree computed in Phase 1 but this time it is different because we will be computing from top-to-bottom. At the top level, we compute $N(\bmod N^2)$. Then the next one, we compute

$$N(\bmod(N_1 \cdot N_2 \cdot \ldots \cdot N_{\frac{m}{2}})^2) \tag{6}$$

and

$$N(\bmod(N_{\frac{m}{2}+1} \cdot N_{\frac{m}{2}+2} \cdot \ldots \cdot N_m)^2). \tag{7}$$

We use the results from the previous level at each following level, and at every level down the tree, the length of numbers is halved. This method is called the remainder tree. For a clearer picture, an illustration by Mironov (2012) is used as an example of a product tree (Figure 1) and the remainder tree (Figure 2) with eight numbers, respectively.

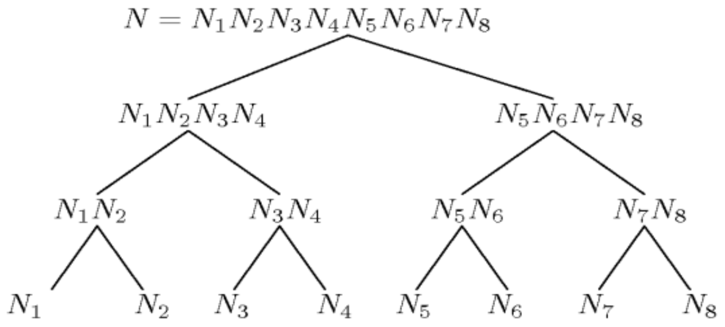


Figure 1: Product tree

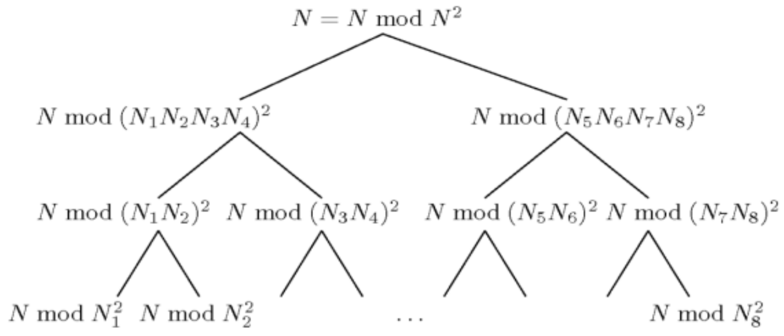


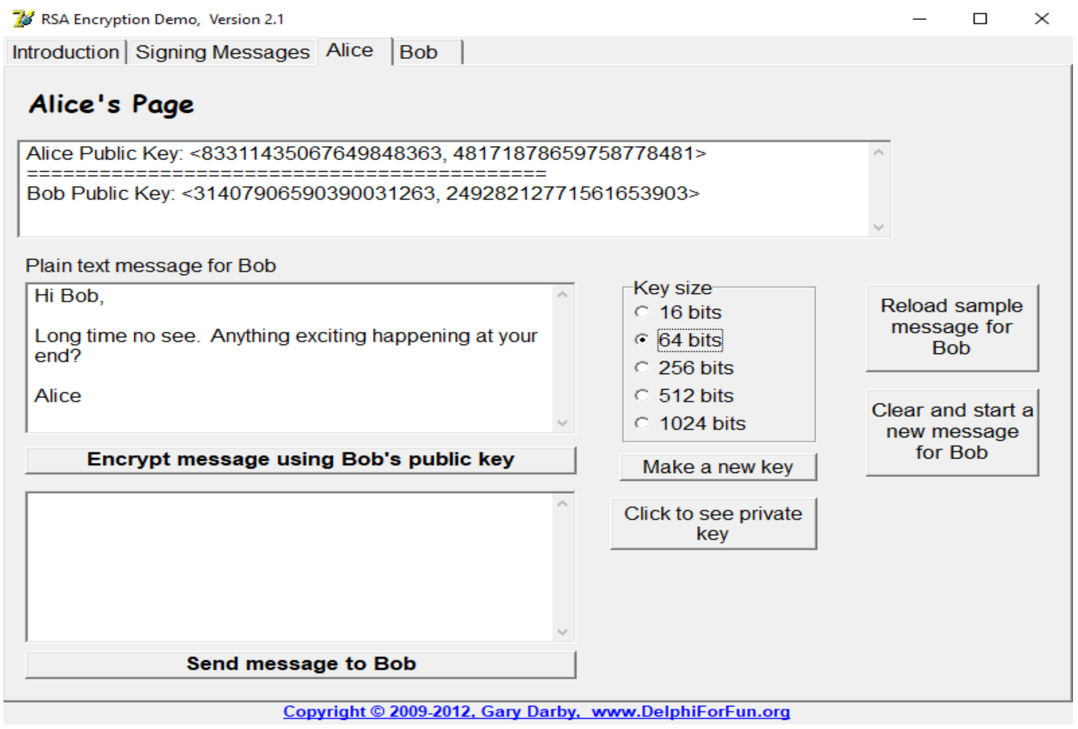
Figure 2: Remainder tree

Phase 3: Computing gcd after the remainder tree is constructed.

According to Cloostermans (2012), after we have obtained information from the remainder tree, we may compute $G_i = \gcd(N_i^2, N(\bmod N_i^2))$ and for most i's, $G_i = N_i$ which means that N_i is co-prime with all other moduli in the dataset. In some cases, we may factor N_i right away if $N_i < G_i < N_i^2$ and, for some case $\gcd\left(\frac{G_i}{N_i}, N_i\right)$ is non-trivial. It means that such computation will cover N_i's whose prime factors are shared with another modulus.

RESULTS AND DISCUSSION

In this section, we will apply the method discussed in the previous section to demonstrate how it works. We will use the RSADemo2 (v2.1) simulator to randomly collect samples of RSA public keys generated and simulate the batch-gcd algorithm using Maple software. Figure 3 is the image of the application, and using this simulator, we can choose between five key lengths (in bits): 16, 64, 256, 512, and 1024.


RSA Encryption Demo, Version 2.1
Introduction
Signing Messages
Alice
Bob
Reload sample message for Bob
Clear and start a new message for Bob
Encrypt message using Bob's public key
Make a new key
Click to see private key
Send message to Bob
Copyright © 2009-2012, Gary Darby, www.DelphiForFun.org

Alice's Page

Alice Public Key: <83311435067649848363, 48171878659758778481>
===
Bob Public Key: <31407906590390031263, 24928212771561653903>

Plain text message for Bob

Hi Bob,

Long time no see. Anything exciting happening at your end?

Alice

Key size
- 16 bits
- [✓ 64 bits]
- 256 bits
- 512 bits
- 1024 bits

Figure 2: RSADemo2 (v2.1) (Gary Darby, DelphiForFun.org)

Experiment with a Small Sample

We will first use small samples to determine whether we can factor our RSA public keys using the batch gcd algorithm. Table 1 shows eight 16-bit RSA keys that we have collected from RSADemo2 (v2.1).

i	1	2	3	4	5	6	7	8
N_i	567109	108601	213013	713809	110893	254477	239011	131981

Table 1. Sampe of RSA moduli generated by RSADemo2 (v2.1)

After gathering samples, we will begin the first phase, which involves utilising a product tree to calculate the product of all moduli. Shown below in Figure 3 is the example of using the product tree to compute the product of all moduli.

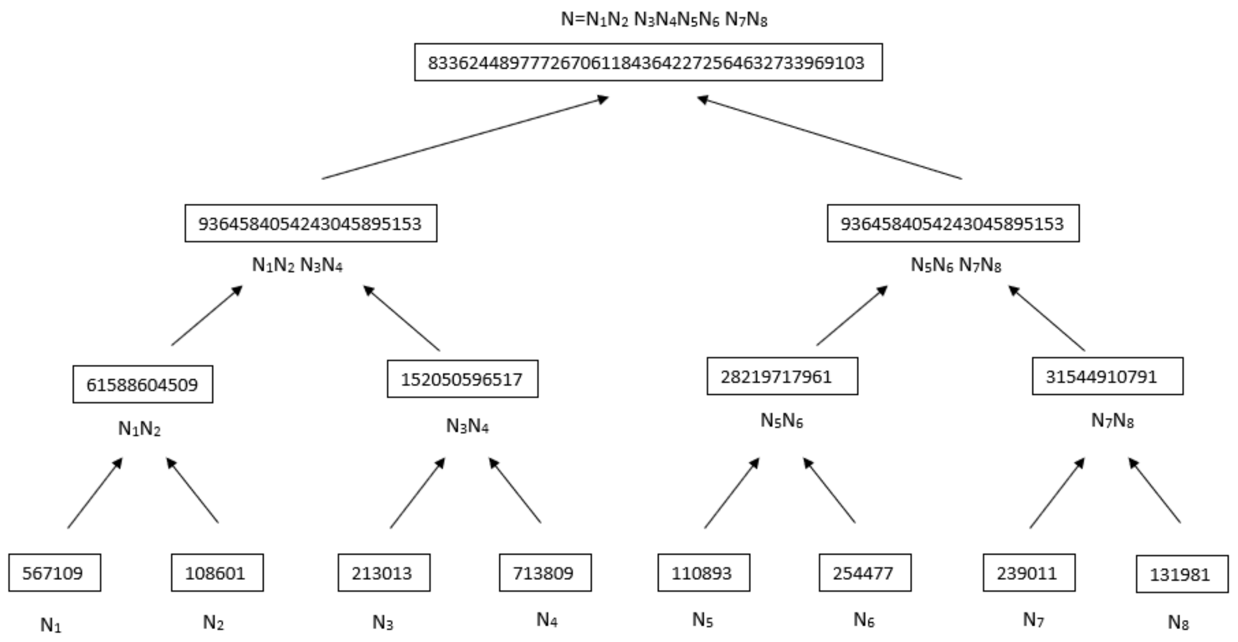


Figure 3: Product tree of sample from Table 1.

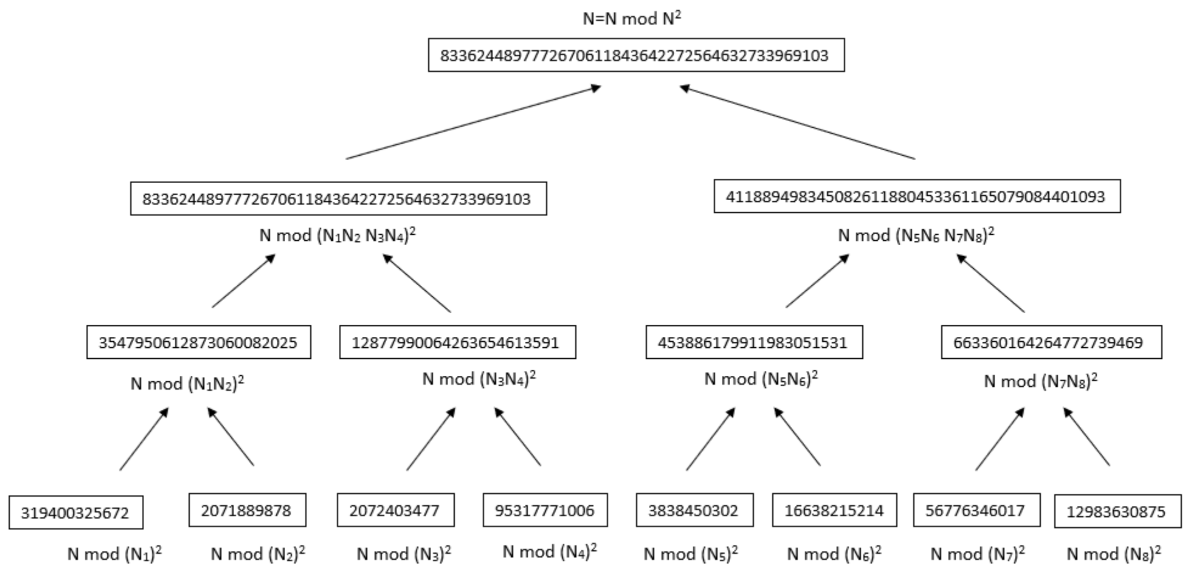


Figure 4: Remainder tree of sample from Table 1.

i	1	2	3	4	5	6	7	8
$N\left(\bmod N_i^2\right)$	319400325672	2071889878	2072403477	95317771006	3838450302	16638215214	56776346017	12983630875

Table 2. List of calculated values $N\left(\bmod N_i^2\right)$

i	1	2	3	4	5	6	7	8
G_i	567109	108601	213013	713809	71082413	163119757	239011	131981
$\gcd\left(\frac{G_i}{N_i}, N_i\right)$	1	1	1	1	641	641	1	1

Table 3. List of factored N_i

Therefore, we have obtained the product of all moduli N where $N = 833624489777267061184364227256463273396 9103$. After that, we will continue with **Phase 2** which we will apply the remainder tree to obtain $N\left(\bmod N_i^2\right)$ as shown in Figure 4 using the same tree as the product tree, but this time, we calculate from the top until the bottom. Hence the list of calculated values $N\left(\bmod N_i^2\right)$ is as presented in Table 2 as follows. From Table 2, we can use the information to compute $G_i = \gcd\left(N_i^2, N(\bmod N_i^2)\right)$.

By using the value of G_i, we can factor N_i. We can see that when the value of $G_i = N_i$, they have one as their only common factor. Then, if $N_i < G_i < N_i^2$, we can factor right away by computing $\gcd\left(\frac{G_i}{N_i}, N_i\right)$. From Table 3 above, we can factor the moduli N_5 and N_6, showing that they share the same prime integers.

Now that we can factor RSA moduli for any bit size using the batch-gcd algorithm by repeating the same step of Phase 1, Phase 2, and Phase 3 until we complete the computation based on Lemma 1. The larger the sample size, the longer it will take to factor all RSA moduli. Furthermore, a problem was encountered during the computation process. We also conducted experiments with 50 samples and not more due to limited computational power. It is due to executing 256-bit RSA moduli. That is why most websites use larger n-bit numbers, making factoring such integers harder and more time-consuming.

CONCLUSION

To conclude this research, this study mainly discusses the most efficient way to factor pair-wise RSA moduli using batch-gcd computation. This method is way more efficient than direct gcd computation based on the Euclidean algorithm. In a further study, we could use high computational power to boost the number of samples we can collect and reduce the time taken to factor all RSA moduli. We could also suggest that the developers use higher n-bit keys to enhance security for users so that their confidential information remains secure and hidden.

REFERENCES

Bernstein, D. J., CDAhang, Y. A., Cheng, C. M., Chou, L. P., Heninger, N., Lange, T., & Van Someren, N. (2013). Factoring RSA keys from certified smart cards: Coppersmith in the wild. In *Advances in Cryptology-ASIACRYPT 2013: 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part II 19* (pp. 341-360). Springer Berlin Heidelberg.

Cloostermans, B. (2012). *Quasi-linear GCD computation and factoring RSA moduli* (Doctoral dissertation, Thesis (Eindhoven University of Technology, Department of Mathematics and Computer Science, 2012)).

Darby, G. (2016). RSA Public Key Demo Program, http://delphiforfun.org/programs/math topics/RSA KeyDemo.htm.

Diffie, W., & Hellman, M. E. (2022). New directions in cryptography. In *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman* (pp. 365-390).

Diffie, W., & Hellman, M. E. (1976). New directions in cryptography. *IEEE transactions on Information Theory*, *22*(6), 644-654.

Heninger, N., Durumeric, Z., Wustrow, E., & Halderman, J. A. (2012). Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *21st USENIX Security Symposium (USENIX Security 12)* (pp. 205-220).

Heninger, N., & Shacham, H. (2009). Reconstructing RSA private keys from random key bits. In *Annual International Cryptology Conference* (pp. 1-17). Berlin, Heidelberg: Springer Berlin Heidelberg.

Kraft, J., & Washington, L. (2018). *An introduction to number theory with cryptography*. CRC press.

Lenstra, A. K., Hughes, J. P., Augier, M., Bos, J. W., Kleinjung, T., & Wachter, C. (2012). Public keys. In *Annual Cryptology Conference* (pp. 626-642). Berlin, Heidelberg: Springer Berlin Heidelberg.

Milanov, E. (2009). The RSA algorithm. *RSA laboratories*, 1-11.

Mironov, I. 2012, Factoring RSA Moduli. Part I., https://windowsontheory.org/2012/05/15/979/.

Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, *21*(2), 120-126.