Containerized Secure Multiparty Computing (MPC) Module for RSA Keypair Generation, Encryption and Decryption

(RSA 키생성 및 암/복호화를 위한 컨테이너화된 안전한 다자간 계산 모듈)

지도교수 : Bernard Egger

이 보고서를 공학학사 학위 논문
대체 보고서로 제출함.

2022 년 6 월 16 일

서울대학교 자연과학대학
생명과학부
Minghang Li

2022 년 8 월

Containerized Secure Multiparty Computing (MPC) Module for RSA Keypair Generation, Encryption and Decryption

(RSA 키생성 및 암/복호화를 위한 컨테이너화된 안전한 다자간 계산 모듈)

지도교수 : Bernard Egger

이 보고서를 공학학사 학위 논문
대체 보고서로 제출함.

2022 년 6 월 16 일

서울대학교 자연과학대학
생명과학부
Minghang Li

2022 년 8 월

Abstract

Secure Multiparty Computing (MPC) is a heated research field in cryptography with the goal of creating methods for multiple parties to jointly contribute to the computation while keeping the input private to each party. Rives-Shamir-Adleman (RSA) encryption algorithm, which requires lots of computations involving multiplication and modulo on large prime numbers, is suitable to be modified to work in an MPC scenario. However, there is almost no existing implementation for distributed RSA keypair generation, and their usage is also limited by the complicated configuration.

Here we present a modern containerized MPC module for RSA keypair generation and decryption. It implements the classic Ben-Or, Goldwasser and Wigderson (BGW) protocol in a highly parallel manner using `gRPC`, a high performance Remote Procedure Call (RPC) framework. The implementation achieved the goal of eliminating the need for trusted dealer in secret sharing and successfully demonstrated the effectiveness of shared RSA key generation. With the sieving method and several pruning techniques applied, it also showed sufficiently high performance, which is about 50 times faster than the traditional single-threaded scheme.

keywords: Cryptography, Secure Multiparty Computing (MPC), Distributed RSA algorithm, Distributed computing, Containerization

Contents

1 Introduction

1.1 Rivest-Shamir-Adleman (RSA) Algorithm

Encryption is core to data privacy and secure communication in the world of Computer Science. In year 1976, Whitfield Diffie and Martin Hellman proposed the first asymmetric encryption algorithm [1], which makes it possible to complete the decryption without directly exchanging the private key between the participants. An asymmetric algorithm usually generates a public key to be distributed openly to others and a private key kept secret to the owner. If the information encrypted by the public key can only be decrypted by the private key, then as long as the private key is not leaked, the communication is secure. This clever design has inspired many scientists to propose novel asymmetric encryption algorithms. In year 1977, Rivest, Shamir and Adleman proposed the Rivest-Shamir-Adleman (RSA) algorithm [2], which is one of the most classic and widely-used asymmetric encryption algorithms up until now.

The RSA algorithm generates the public key and the private key as follows:

1. Select a prime number p and a prime number q.
2. Compute $n = pq$ and $\phi = (p-1)(q-1)$.
3. Select an integer e such that $1 \leq e < \phi$ and $\gcd(e, phi) = 1$.
4. Compute $d = e^{-1} \mod \phi$.

As a result, the public key is $\{e, n\}$ and the private key is $\{d, n\}$. To encrypt a message M with the public key, we use the following formula:

$$C = M^e \bmod n \tag{1.1}$$

Vice versa, to decrypt a message C with the private key, we use the following formula:

$$M = C^d \bmod n \tag{1.2}$$

1.2 Secure Multiparty Computing (MPC)

Secure Multiparty Computing (MPC) is a way of enabling a group of data owners to jointly compute a function using all their data as inputs, without disclosing any participant's private input to each other or any third party [3]. This idea was first introduced in Yao's discussion on the famous Yao's millionaire problem in the early 1980s [4]. He also raised the first MPC protocol: the Garbled Circuits (GC) Protocol [4], which remains the basis for many current MPC implementations. From then on, many MPC protocols have been proposed, such as the Goldreich-Micali-Wigderson (GMW) protocol [5], the Ben-Or-Goldwasser-Wigderson (BGW) [6] protocol and GESS [7] protocol. Some of the protocols are designed for only two participants (2-party MPC or 2-PC) or three participants (3-PC), while others can be generalized to many participants. The security level of the protocols also varies: all the protocols listed previously are secure only for *semi-honest* adversaries who follow the protocol as specified but may try to learn as much as possible from the message they received [3]. For higher security against *malicious* adversaries who may deviate from what the protocol required, novel protocols like Cut-and-Choose [8] and GMW compiler [5] protocols were proposed.

1.3 RSA Algorithm under MPC Scenario

It is possible and suitable to convert RSA algorithm to be a distributed protocol using MPC. Under an MPC scenario, the modulus N and the private key exponent d are split into several parts and shared among the participants. To be specific, instead of directly disclosing $N = pq$ to k participants, we define

$$N = \left(\sum_{1}^{k} p_i\right)\left(\sum_{1}^{k} q_i\right) \tag{1.3}$$

where p_i and q_i are only known to each individual participant i. Similarly the private exponent d is also defined by

$$d = \sum_{1}^{k} d_i \tag{1.4}$$

where each participant i only knows d_i instead of the full d. While using the distributed private key to decode message C, each participant i only computes a partial decryption (terms *shadow*):

$$s_i = C^{d_i} \mod N \tag{1.5}$$

The full result can be obtained by the product of all shadows:

$$M = \prod_{1}^{k} s_i = C^{\sum d_i} \mod N = C^d \mod N \tag{1.6}$$

The shared modulus generation and shared private key generation is a suitable scenario to apply the BGW protocol. The method of implementation using BGW protocol has been raised by Boneh and Franklin in year 1997 (referred to as BF97)[9]. Although the algorithm is proposed several years ago, there is no recent usable implementation of it.

1.4 Background Study of Existing MPC Applications

Apart from the primitives and software only for academical uses (e.g., verifying the correctness of the theorems in the paper), there are still many secure multiparty computing frameworks focusing on different techniques and protocols to provide secure computation. However, most of them only support 2PC (two-party computing) or 3PC (three-party computing); some of them are tightly integrated in other frameworks, like `CrypTen`, a MPC library focusing only on building `PyTorch` applications [10]; some of them focus on non-suitable protocols for RSA keypair generation, such as gabled circuits and zero knowledge proof, like `FRESCO` [11] and `EMP-toolkit` [12]; some of them are very difficult to compile and use like `SCALE-MAMBA` (a work based on Bendlin et. al.[13], Damgard et. al.[14], Nielsen et. al.[15] and Hazay et. al.[16]). To sum up, no suitable product has come into market providing a reliable, easy-to-use, dedicated secure multiparty computation application for RSA keypair generation and decryption. Our application aims to fill in such a niche.

1.5 Motivation

As is analyzed in the previous sections, The goal and major contribution described in this report is the implementation of a multiparty computing application for RSA key generation with containerization support for easy deployment.

1.6 Division of Work

This course project was implemented by Minghang Li, Hexiang Geng and Gyeongjun Lee. Minghang Li and Hexiang Geng designed and implemented the overall `gRPC` master-worker framework. The modulus generation, primality test and Doral private key sharing functions called by the `gRPC` services were also implemented by Minghang Li. The decryption function and the support for RSA+ECB+PKCS1 standard was implemented by Gyeongjun Lee. This report will focus on describing the optimizations done by Minghang Li and will only mention briefly the most general abstractions of the other team members' work in order to describe the contribution of Minghang Li.

2 Materials & Methods

2.1 Keypair Generation Algorithm

The process is further divided into modulus generation, primality test and key generation.

2.1.1 Modulus Generation

The goal is to generate a modulus $N = \left(\sum_i p_i\right)\left(\sum_i q_i\right)$, where p_i and q_i are private to the participant.

A prime number P satisfying $P > N$ is sent to all workers.

Let $l = \left\lfloor \frac{k-1}{2} \right\rfloor$, $\forall i = 1, ..., k$, worker i picks two random degree l polynomials $f_i, g_i \in \mathbb{Z}_P[x]$ satisfying $f_i(0) = p_i$, $g_i(0) = q_i$, and a random degree $2l$ polynomial $h_i \in \mathbb{Z}_P[x]$ satisfying $h_i(0) = 0$.

For all $i = 1, ..., k$, each worker i computes:

$$\begin{aligned} \forall j = 1, ..., k, \quad p_{i,j} &= f_i(j) \\ q_{i,j} &= g_i(j) \\ h_{i,j} &= h_i(j) \end{aligned} \tag{2.1}$$

Worker i then privately sends the tuple $\langle p_{i,j}, q_{i,j}, h_{i,j} \rangle$ to server j for all $j \neq i$.

After the communication, each worker i has all of $\langle p_{i,j}, q_{i,j}, h_{i,j} \rangle, \quad \forall j = 1, ..., k$. Worker i then computes:

$$N_i = \left(\sum_{j=1}^{k} p_{j,i}\right)\left(\sum_{j=1}^{k} q_{j,i}\right) + \sum_{j=1}^{k} h_{j,1} \pmod{P} \tag{2.2}$$

After the computation, N_i is broadcast to all other workers. Now, each worker j has all values of N_i for $i = 1, ..., k$.

Let $\alpha(x)$ be the polynomial

$$\alpha(x) = \left(\sum_j f_j(x)\right)\left(\sum_j g_j(x)\right) + \sum_j h_j(x) \pmod{P} \tag{2.3}$$

Observe that $\alpha(i) = N_i$ and by definition of f_i, g_i and h_i, we have $\alpha(0) = N$. Furthermore, $\alpha(x)$ is a polynomial of degree $2l$. We note that l is defined in such at way so that $k \leq 2l + 1$. Since all servers have at least $2l + 1$ points on $\alpha(x)$, they can interpolate the points to obtain the coefficients of $\alpha(x)$. Each server finally evaluates $\alpha(0)$ using the interpolated coefficients and obtains $N \mod P$. Since $N < P$, each server learns the correct value of N without disclosing any information.

2.1.2 Primality Test of Modulus

A random number $g \in \mathbb{Z}_N^*$ is chosen and broadcast to all k workers.

Worker 1 (not necessarily the first worker, just selected to be the "primary" worker) computes:

$$v_1 = g^{N-p_1-q_1+1} \tag{2.4}$$

All other workers compute

$$v_i = g^{p_i+q_i} \tag{2.5}$$

The workers then exchange the v_i values with each other and verify that

$$v_1 = \prod_{i=2}^{k} v_i \pmod N \tag{2.6}$$

The test is referred as Fermat test for testing that a number is a product of two primes. It is possible that for a N that is not a product of two primes to pass the test, but the density is significantly small (less than 1 out of 10^{40}).

2.1.3 Shared Generation of Private Keys

For distributed generation of private keys, we use the method introduced by Dario Catalano, Rosario Gennaro and Shai Halevi for its simplicity and security under the assumed condition we set.

A number e is chosen and known to all workers.

Worker 1 (not necessarily the first worker, just selected to be the "primary" worker) locally computes:

$$\phi_1 = N - p_1 - q_1 + 1 \tag{2.7}$$

All other workers copmute

$$\phi_i = -p_i - q_i \tag{2.8}$$

It's obvious that

$$N = \sum_i \phi_i, \quad i = 1, ..., k \tag{2.9}$$

Then, each worker computes

$$\gamma_i = r_i \cdot e + \phi_i \tag{2.10}$$

where r_i is a random number decided by the worker. γ_i is then shared among all workers. And then all workers can compute:

$$\gamma = \sum_i \gamma_i = \phi + R \cdot e \tag{2.11}$$

There are a and b such that

$$a \cdot \gamma + b \cdot e = 1, \qquad d = a \cdot R + b \tag{2.12}$$

Then we can compute

$$a = \gamma^{-1} \pmod e, \qquad b = \frac{1 - a \cdot \gamma}{e} \tag{2.13}$$

With all the information each worker holds, a usable additive share of d can be:

$$d_1 = a \cdot r_1 + b, \qquad d_{i \neq 1} = a \cdot r_i \tag{2.14}$$

2.2 Optimizations

Many optimizations are applied in various aspects in this project, but the contributions of most of them are either difficult to measure or optimized for better memory consumption or network bandwidth consumption. The most conspicuous and important optimizations in the aspect of speeding up the running speed of this application are the two listed below: distributed sieving and parallel modulus generation.

2.2.1 Distributed Sieving Algorithm

In Section 2.1.1, we set p_i and q_i to be random prime numbers and use a primality test to verify whether $N = \left(\sum_1^k p_i\right)\left(\sum_1^k q_i\right)$ only has two prime factors. As the generated key bit length increases, the possibility that N satisfies our requirement decreases quadratically, making this method unpractical for real-world application.

Distributed Sieving is an important optimization technique to eliminate small prime factors from the generated N by choosing p_i and q_i wisely.

When the sieving started, a serires of small prime number is generated, and $M < 2^{\text{factor length}} - 1$ is computed to be the product of all the prime numbers. Each worker then chooses a_i that is coprime to M. Note that the product of all a_i, A:

$$A = \prod_{i=1}^{k} a_i \tag{2.15}$$

is also coprime to M. We term a_i as a *multiplicative* share of A.

Suppose we have some b_i such that:

$$A \pmod{M} = \sum_{i=1}^{k} b_i \pmod{M} \tag{2.16}$$

Suppose we let this b_i serve as p_i, it is clear that:

$$p = \sum_{i=1}^{k} p_i = \sum_{i=1}^{k} b_i \tag{2.17}$$

is also coprime to M. We term b_i as the additive share of A.

Hence, if each worker selects an a_i coprime to M, and the product of all a_i, a is converted from a multiplicative share to an additive share securely, we'll be able to obtain a p_i (and q_i using similar methods) whose sum is coprime to M. The next step is to use a proper method to perform the conversion.

It would be beneficial to consider the conversion process step by step, one worker at a time. For some $1 \le l < k$, the value A_l is defined to be the product of value a_1 to a_l :

$$\begin{aligned} A_l &= \prod_{i=1}^{l} a_i \pmod{M} \\ &= a_1 \cdots a_l \pmod{M} \end{aligned} \tag{2.18}$$

Suppose that a_l is already converted to an additive sharing:

$$\begin{aligned} A_l &= \sum_{i=1}^{l} b_{i,l} \pmod{M} \\ &= b_{1,l} + \cdots + b_{k,l} \pmod{M} \end{aligned} \tag{2.19}$$

It is possible for us to use the BGW protocol to further convert the value A_{l+1} into additive sharing. Recall the implementation of BGW protocol in Section 2.1.1, according to Lagrange interpolation we know that:

$$N = \alpha(0) = \sum_{i=1}^{k} \lambda_i(0) N_i \pmod{P} \tag{2.20}$$

where $\lambda_i(0)$ is the Lagrange interpolation coefficient defined by:

$$\lambda_i(x) = \prod_{j \ne i} \frac{x-j}{i-j} \tag{2.21}$$

If, instead of broadcasting N_i, each worker preserves the value M_i, which is defined as:

$$M_i = \lambda_i(0) N_i \pmod{P} \tag{2.22}$$

We can see that in essence we create an additive sharing of the two series $\sum p_i$ and $\sum q_i$. That is to say:

$$\sum_{i=1}^{k} M_i = N = \left(\sum_{i=1}^{k} p_i\right)\left(\sum_{i=1}^{k} q_i\right) \tag{2.23}$$

Given the previous assumption, it's possible to re-write the above equation of a_{l+1} as:

$$
\begin{aligned}
A_{l+1} &= \prod_{i=1}^{l+1} a_i \pmod M \\
&= a_{l+1} \cdot \prod_{i=1}^{l} a_i \pmod M \\
&= a_{l+1} \cdot \sum_{i=1}^{k} b_{i,l} \pmod M
\end{aligned} \tag{2.24}
$$

Then, if we can devise a series $\{u_i\}$ such that:

$$
a_{l+1} = \sum_{i=1}^{k} u_i \tag{2.25}
$$

Since BGW protocol can be used to share the product of two sums, we can apply it to safely share A_{l+1} between the participants:

$$
A_{l+1} = \sum_{i=1}^{k} u_i \sum_{i=1}^{k} b_{i,l} \tag{2.26}
$$

The series $\{u_i\}$ described below trivially satisfies the requirement:

$$
\begin{cases}
u_i = a_i, & \text{if } i = l+1 \\
u_i = 0, & \text{if } i \neq l+1
\end{cases} \tag{2.27}
$$

Starting from $l = 1$, after $k - 1$ iterations of this procedure, we will obtain the desired additive sharing of A.

Going back to the application in the current application, to apply this procedure, we only need to let worker 1 computes a series $\{b_{1,j}\}$ with k elemnets, such that:

$$
a_1 = \sum_{j=1}^{k} b_{1,j} \tag{2.28}
$$

Next, worker 1 sends $b_{1,j}$ to the corresponding worker $j = 2, 3, \ldots, k$. And then the procedure was repeated for $k - 1$ times by the following workers to produce the desired additive sharing.

After the procedure is finished, each worker i picks a random number r_i in range $\left[0, \frac{2^n}{M}\right]$ as sets

p_i to be:

$$p_i = r_i M + b_i \tag{2.29}$$

Clearly, the following relationship holds:

$$p = \sum_{i=1}^{k} p_i \equiv A \mod M \tag{2.30}$$

Hence, p is not divisible by all the prime factors of M. Similarly, we can apply the same technique for the selection of q_i. This pruning will drastically decrease the generation time for the modulus.

2.2.2 Parallel Modulus Generation

The process of modulus generation is a randomized process, even with the help of sieving, it's not guaranteed how many attempts will be gone through before a valid modulus N is generated.

During the generation, the cluster faces many synchronization points where some workers must stop computing and wait for other ones to complete the previous procedure, leaving some of the CPU threads idle or underutilized. The unbalanced work assignment such as in the primality test, worker 1 is taking significantly more computation job than other workers, makes this problem even worse. A simple work balancing will not help since the cluster always needs to wait for one test holder to complete result checking.

Thus, generating modulus and testing in parallel is of great help. In our implementation, when a parallel flag is given, each worker will host an independent modulus generation-test loop until one of them has found a valid N , after which all other workers will abort the hosted job and participate the private key generation using the valid N.

To eliminate the interference of parallel workflow, each workflow has a unique workflow ID attached when sending messages to other workers. And the data receiving and waiting mechanism is also optimized for this feature.

Theoretically this optimization will massively increase CPU utilization and although it may potentially decrease the speed of each generation-test workflow, it will prevent the relatively bad cases and thus increase the overall performance. But the final effect differs by the cluster size, worker specs and connection quality.

2.3 Implementation

The application has two different working modes; thus, the architecture differs with different mode. In worker mode, the application contains two major modules, worker RPC service and worker control module. The worker RPC service manages the communication between other RPC services and is used to form the computation cluster. The worker control module contains the computation logic for key generation and message decryption. Since the worker process does not need to read user console input aside from the starting command line arguments, the worker control module does have any input function. In manager mode, the application contains two major modules, manager RPC service and manager control logic. The manager RPC service communicates to all other RPC services to form the cluster. The manager control module needs to interact with user through command line input to gain information such as worker URIs.

A cluster includes at least 3 worker nodes and at least 1 manager node. The type of node is determined by the command line argument sent to the application and a client can start work as a worker and a manager simultaneously. The application will start a worker node and/or a manager node and enter the interactive mode of a manager or waiting for a worker to terminate accordingly. The components can be easily transplanted and used in other projects and work in non-interactive mode.

Each worker includes 4 components, the control logic, the RPC sender, the RPC receiver and the data receiver. RPC sender sends RPC request to other workers. Due to the life span of an outgoing RPC call depends on the sending thread' s, an async executor pool for sending requests and optimize it to reduce delays. The RPC receiver receives incoming RPC requests and handles the request to the data receiver or the control logic depending on the RPC type. The workflow is highly parallel, and the information needed for the computation is shared via push messages, thus data receiver uses concurrent maps to store information across different parallel workflows.

Manager only has 3 components, the RPC sender, control logic and data receiver. The way they function is very similar to those of a worker, but a manager does not have an RPC server as it only sends requests to workers.

The example clusters we are showing are made up with a single manager and multiple workers, but the number can vary, and our app also supports a client to run in both manager mode and worker mode simultaneously.

The application has a command line user interface and will start at different mode with different starting arguments given. User needs to assign port number for communication and the IP address needs to be input from console for a process in manager mode.

The internal implementation of `WorkerMain` and `ManagerMain` supports to be used in a non-interactive mode with other code, but the application wrapped around them will always start a manager in interactive mode for demonstration purpose.

In each process, different functional modules are different working phases of one single process, thus no communication interface is needed. Between processes, the communication is managed by gRPC modules. gRPC is used for defining, sending, and receiving messages, with blocking and unblocking ways.

3 Results

In this chapter, the correctness and efficiency of the full algorithm and the effect of the optimizations were shown. The benchmark platforms are 3 personal computers with slightly heterogeneous hardware and operating systems. The following list showed the hardware and operating systems of the benchmark platforms.

1. **PC1**: AMD Ryzen 7 4800U, 16GB RAM, Windows 10; running 2 workers
2. **PC2**: AMD Ryzen 7 1700, 16GB RAM, Windows 10; running 2 workers
3. **PC3**: AMD Ryzen 7 2700X, 32GB RAM, Windows 10; running 1 workers

During testing, the cluster can generate a public key and corresponding distributed private keys correctly and efficiently. The generated public key can be printed in correct format and the information can be parsed correctly using 3rd party key parser. A message can be encrypted by the manager using the implemented encryption algorithm, and the encrypted message can be correctly decrypted by the workers using the distributed private keys.

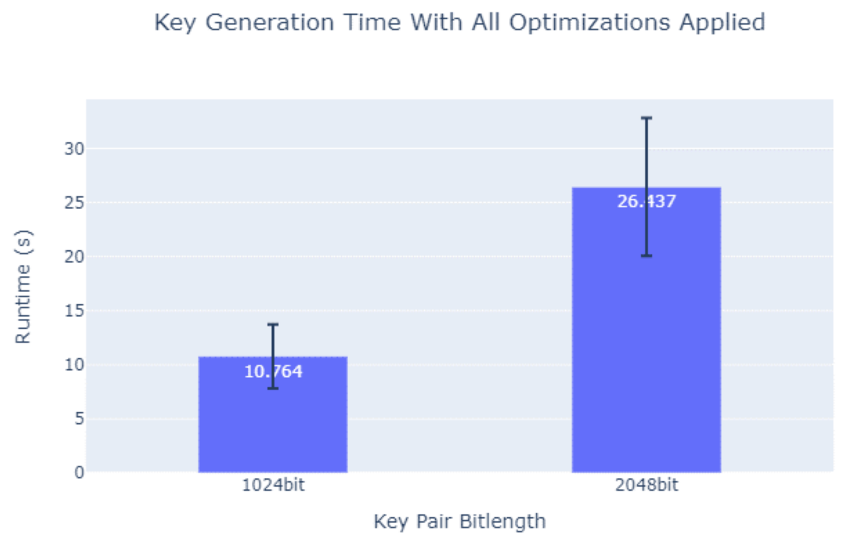

Figure 3.1: This figure shows the time required for key pair generation for keys with different bit length. On the tested platforms, the total generation time for a 1024-bit key is only 10 seconds, whereas the total generation time for a 2048-bit key is about 26 seconds.

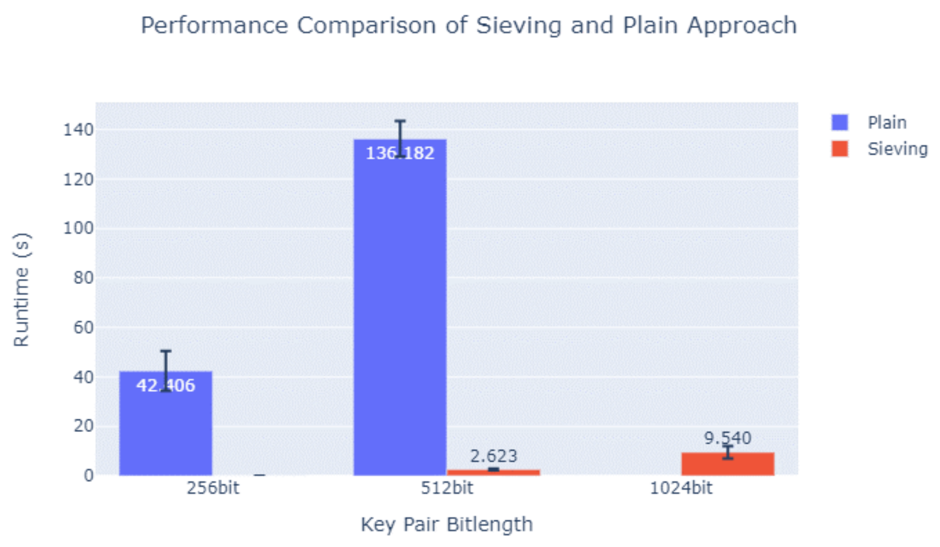

Figure 3.2: This figure shows the effect of applying distributed sieving. Without applying distributed sieving, generation a merely 256-bit long RSA key pair took already 42 seconds, which is already slower than genearting a 4 times longer key pair with distributed sieving enabled. Generating a 512-bit key pair took more than 1.5 minutes, whereas the optimized version took no more than 3 seconds.

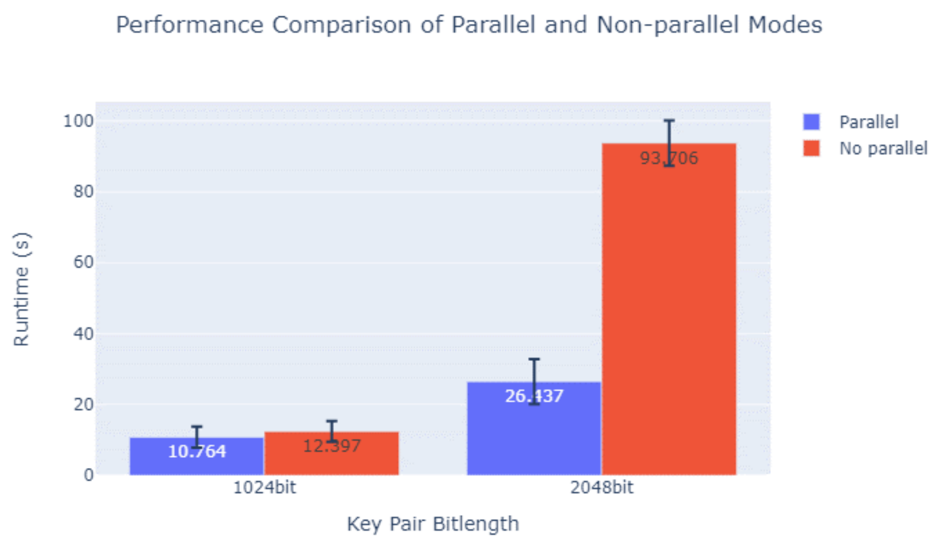

Figure 3.3: This figure shows the effect of parallelization. The effect is conspicuous for 2048-bit long RSA keys. Without enabling parallelization, the generation of a 2048-bit long RSA key pair took about 1.5 minutes, whereas the optimized version took only half a minute.

4 Conclusion

In conclusion, our application works as expected. We' ve successfully implemented a Secure Multiparty Computation framework for distributed RSA keypair generation and decryption. It is overall easy to use and has good performance.

The best application scenario of our work is multiple workers helping one manager to decrypt an incoming encrypted message. owever, although the distributed public key and private key generation is achieved and a message encrypted by the manager can be distributed decrypted successfully, the compatibility with other encryption protocols is not fully implemented. The encryption/decryption protocol we implemented is using the RSA + ECB + PKCS1 standard. However, the encryption protocols used for secure message transmission is more complicated and requires more processing. To support so we may continue to expand our encryption/decryption algorithm or turn to use existing open-source libraries.

Reference

[1] W. Diffie and M. Hellman, “New directions in cryptography,” *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, Nov. 1976, Conference Name: IEEE Transactions on Information Theory, ISSN: 1557-9654. DOI: `10.1109/TIT.1976.1055638`.

[2] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978, ISSN: 0001-0782. DOI: `10.1145/359340.359342`. [Online]. Available: `https://doi.org/10.1145/359340.359342` (visited on 06/15/2022).

[3] D. Evans, V. Kolesnikov, and M. Rosulek, “A Pragmatic Introduction to Secure Multi-Party Computation,” English, *SEC*, vol. 2, no. 2-3, pp. 70–246, Dec. 2018, Publisher: Now Publishers, Inc., ISSN: 2474-1558, 2474-1566. DOI: `10.1561/3300000019`. [Online]. Available: `https://www.nowpublishers.com/article/Details/SEC-019` (visited on 06/15/2022).

[4] A. C. Yao, “Protocols for secure computations,” in *23rd Annual Symposium on Foundations of Computer Science* (*sfcs 1982*), ISSN: 0272-5428, Nov. 1982, pp. 160–164. DOI: `10.1109/SFCS.1982.38`.

[5] O. Goldreich, S. Micali, and A. Wigderson, “How to play any mental game, or a completeness theorem for protocols with honest majority,” in *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 307–328, ISBN: 978-1-4503-7266-4. [Online]. Available: `https://doi.org/10.1145/3335741.3335755` (visited on 06/14/2022).

[6] M. Ben-Or, S. Goldwasser, and A. Wigderson, “Completeness theorems for non-cryptographic fault-tolerant distributed computation,” in *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 351–371, ISBN: 978-1-4503-7266-4. [Online]. Available: `https://doi.org/10.1145/3335741.3335756` (visited on 06/14/2022).

[7] V. Kolesnikov, "Gate Evaluation Secret Sharing and Secure One-Round Two-Party Computation," en, in *Advances in Cryptology - ASIACRYPT 2005*, B. Roy, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2005, pp. 136–155, ISBN: 978-3-540-32267-2. DOI: `10.1007/11593447_8`.

[8] D. Chaum, "Blind Signature System," en, in *Advances in Cryptology: Proceedings of Crypto 83*, D. Chaum, Ed., Boston, MA: Springer US, 1984, pp. 153–153, ISBN: 978-1-4684-4730-9. DOI: `10.1007/978-1-4684-4730-9_14`. [Online]. Available: `https://doi.org/10.1007/978-1-4684-4730-9_14` (visited on 06/15/2022).

[9] D. Boneh and M. Franklin, "Efficient generation of shared RSA keys," en, in *Advances in Cryptology — CRYPTO '97*, B. S. Kaliski, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 1997, pp. 425–439, ISBN: 978-3-540-69528-8. DOI: `10.1007/BFb0052253`.

[10] B. Knott, S. Venkataraman, A. Hannun, S. Sengupta, M. Ibrahim, and L. van der Maaten, "CrypTen: Secure Multi-Party Computation Meets Machine Learning," in *Advances in Neural Information Processing Systems*, vol. 34, Curran Associates, Inc., 2021, pp. 4961–4973. [Online]. Available: `https://proceedings.neurips.cc/paper/2021/hash/2754518221cfbc8d25c13a06a4cb8421-Abstract.html` (visited on 06/15/2022).

[11] *FRESCO - a FRamework for Efficient Secure COmputation*, original-date: 2015-11-24T12:05:17Z, Jun. 2022. [Online]. Available: `https://github.com/aicis/fresco` (visited on 06/15/2022).

[12] X. Wang, A. J. Malozemoff, and J. Katz, *EMP-toolkit: Efficient MultiParty computation toolkit*, `https://github.com/emp-toolkit`, 2016.

[13] R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias, "Semi-Homomorphic Encryption and Multiparty Computation," en, *Cryptology ePrint Archive*, 2010. [Online]. Available: `https://eprint.iacr.org/2010/514` (visited on 06/15/2022).

[14] I. Damgard, V. Pastro, N. P. Smart, and S. Zakarias, "Multiparty Computation from Somewhat Homomorphic Encryption," en, *Cryptology ePrint Archive*, 2011. [Online]. Available: `https://eprint.iacr.org/2011/535` (visited on 06/15/2022).

[15] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra, "A New Approach to Practical Active-Secure Two-Party Computation," en, *Cryptology ePrint Archive*, 2011. [Online]. Available: `https://eprint.iacr.org/2011/091` (visited on 06/15/2022).

[16] C. Hazay, P. Scholl, and E. Soria-Vazquez, "Low Cost Constant Round MPC Combining BMR and Oblivious Transfer," en, *Cryptology ePrint Archive*, 2017. [Online]. Available: `https://eprint.iacr.org/2017/214` (visited on 06/15/2022).

국문초록

RSA 키생성 및 암/복호화를 위한 컨테이너화된 안전한 다자간 계산 모듈

Minghang Li

College of Natural Sciences

Department of Biological Sciences

Seoul National University

MPC(Secure Multiparty Computing)는 입력을 각 당사자에게 비공개로 유지하면서 여러 당사자가 계산에 공동으로 기여할 수 있는 방법을 만드는 것을 목표로 하는 암호화 연구 분야입니다. 큰 소수에 대한 곱셈 및 모듈로와 관련된 많은 계산이 필요한 RSA(Rives-Shamir-Adleman) 암호화 알고리즘은 MPC 시나리오에서 작동하도록 수정하기에 적합합니다. 그러나 분산 RSA 키 쌍 생성을 위한 기존 구현이 거의 없으며 복잡한 구성으로 인해 사용도 제한됩니다.

여기에서는 RSA 키 쌍 생성 및 암호 해독을 위한 최신 컨테이너화된 MPC 모듈을 제공합니다. 고성능 RPC(원격 프로시저 호출) 프레임워크인 `gRPC`를 사용하여 고전적인 Ben-Or, Goldwasser 및 Wigderson(BGW) 프로토콜을 고도의 병렬 방식으로 구현합니다. 구현은 비밀 공유에서 신뢰할 수 있는 딜러의 필요성을 제거한다는 목표를 달성하고 공유 RSA 키 생성의 효율성을 성공적으로 입증했습니다. 체질 방법과 여러 가지 가지치기 기술을 적용하여 기존의 단일 스레드 방식보다 약 50배 더 빠른 성능도 충분히 보여주었습니다.

주요어: 암호화, 안전다방계산, 분산 RSA 알고리즘, 분산 컴퓨팅, 컨테이너화