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 11 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 11 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 modulus on large prime numbers, is suitable to be modified to work in an MPC scenario. However, there is no existing implementation for distributed RSA keypair generation.

Here we present a modern containerized MPC module for RSA keypair generation, encryption and decryption. It implements the classic Boneh & Franklin Scheme 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.

The MPC RSA module is freely open source at *https://github.com/matchy233/mpc-rsa*. The Docker images can be found at `matchy233/mpc-project_manager` and `machy233/mpc-project_worker`.

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

Contents

1 Introduction

1.1 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 [1]. This idea was first introduced in Yao's discussion on the famous Yao's millionaire problem in the early 1980s [2]. He also raised the first MPC protocol: the Garbled Circuits Protocol [2], which remains the basis for many current MPC implementations. From then on, various MPC protocols have been proposed.

(discuss MPC protocols, briefly mention garbled circuits and detail on shared secrets)

1.1.1 Garbled Circuits

1.1.2 Shamir Key Sharing

1.2 Rivest-Shamir-Adleman (RSA) Algorithm

(just brief, focus on the concepts)

1.3 Background Study of Existing MPC Applications

According to our research, 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 CypTen, a MPC library focusing only on building PyTorch applications; some of them focus on non-suitable protocols for RSA keypair generation, such as gabled circuits and zero knowledge proof, like FRESCO and EMP-toolkit; some of them are very difficult to compile and use like SCALE-MAMBA. To sum up, no suitable product has come into market providing a reliable, asy-to-use, dedicated secure multiparty computation application for RSA keypair generation and decryption. Our application aims to fill in such a niche.

1.4 Motivation and Contribution of This Thesis

As is analyzed in the previous sections, The goal and major contribution of this thesis the implementation of a multiparty computing application for RSA key generation and message decryption with containerization support for easier deployment. The process of computation will be t-out-of-k level secure against attackers (the current approach ensures a l-outof-k threshold, see Part E) and the communication will be SSL protected.

2 Materials & Methods

2.1 Keypair Generation Algorithm

Different from a regular RSA key generation process, where $N = pq$, p and q are prime numbers, a distributed modulo has

$$N = \left(\sum_{1}^{k} p_i\right)\left(\sum_{1}^{k} q_i\right) \tag{2.1}$$

During the computation, the value of each p_i and q_i is not explicitly revealed among workers.

A method described by D. Boneh and M. Franklin (citation) is used. The process is further divided into modulo generation, primality test and key generation.

2.1.1 Modulus Generation

One prime number P satisfying $P > N$ is sent to all workeres.

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:

$$\begin{aligned} \forall j = 1, ..., k, \quad p_{i,j} &= f_i(j) \\ q_{i,j} &= g_i(j) \\ h_{i,j} &= h_i(j) \end{aligned} \tag{2.2}$$

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:

$$N_i = \left(\sum_{j=1}^{k} p_{j,i}\right)\left(\sum_{j=1}^{k} q_{j,i}\right) + \sum_{j=1}^{k} h_{j,1}(\mod P) \tag{2.3}$$

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

$$\alpha(x) = \left(\sum_{j} f_j(x)\right)\left(\sum_{j} g_j(x)\right) + \sum_{j} h_j(x)(\mod P) \tag{2.4}$$

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.

Note that in the above process $p_{i,j}$ and $q_{i,j}$ are the standard l-out-of-k Shamir sharing of p_i and q_i.

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:

$$v_1 = g^{N-p_1-q_1+1} \tag{2.5}$$

All other workers compute

$$v_i = g^{p_i+q_i} \tag{2.6}$$

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

$$v_1 = \prod_{i=2}^{k} v_i \pmod N \tag{2.7}$$

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:

$$\Phi_1 = N - p_1 - q_1 + 1 \tag{2.8}$$

All other workers copmute

$$\Phi_i = -p_i - q_i \tag{2.9}$$

It’s obvious that

$$N = \sum_i \Phi_i, \quad i = 1, ..., k \tag{2.10}$$

Then, each worker computes

$$\gamma_i = r_i \cdot e + \Phi_i \tag{2.11}$$

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:

$$\gamma = \sum_i \gamma_i = \Phi + R \cdot e \tag{2.12}$$

There are a and b such that

$$a \cdot \gamma + b \cdot e = 1d = a \cdot R + b \tag{2.13}$$

Then we can compute

$$a = \gamma^{-1} (\mod e), \qquad b = \frac{1 - a \cdot \gamma}{e} \tag{2.14}$$

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

$$d_1 = a \cdot r_1 + b, \qquad d_{i \neq 1} = a \cdot r_i \tag{2.15}$$

2.2 Message Encryption and Decryption Algorithm

In order to communiate with other application of web, we select to use `RSAES-PKCS1-V1_5-ENCRYPT` scheme. This scheme uses both RSA and ECB algorithm. First, we split the message to encrypt into n blocks, and then do RSA encryption to each of it. This scheme includes 11 bytes padding, so preprocessing is required for each block. The first two bytes of the padding are `0x00` and `0x02`, followed by a randomly generateed 8-byte octet string. The last byte of the padding is `0x00`. The original message was attached at the end of the padding.

After preprocessing, we do RSA encryption block by block. Let block of string EM, then the encrypted message C is

$$C = EM^e \mod N \tag{2.16}$$

And we combine C to make the whole encrypted message M.

To decrypt this message M, the worker get the input encrypted message M and computes

$$s_i = m^{d_i} \mod N \tag{2.17}$$

The result (called *shadow*) is sent back to the receiver (manager), and manager computes

$$s = \prod_i s_i = m^{\sum_i d_i} = m^d \mod N \tag{2.18}$$

Thus, the valid decrypted message is obtained.

2.3 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.

2.3.1 Distributed Sieving Algorithm

In section (), 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.

(unfinished)

2.3.2 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 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 , after which all other workers will abort the hosted job and participate the private key generation using the valid .

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.

2.4 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 nodes URIs.

(figure) shows an example cluster. 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 command line argument parsing is implemented by hand and requires the user to give information in a fixed format.

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

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, show generation time w.r.t. keypair bitlength with all optimization applied)

(show sieving effect)

(show parallelization effect)

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.

5 Future Work

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.

Currently the way that a manager takes in encrypted messages is either through the command line in interactive mode or use the decrypt() api provided by the ManagerMain class when using the source code of our project. However, the ideal case should be either the manager works as an https/encrypted RPC server, or our project compiles to a library for easier use with other projects.

Acknowledgements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Appendix

5.1 Run-times of keypair generation time

Reference

[1] D. Evans, V. Kolesnikov, M. Rosulek, *et al.*, "A pragmatic introduction to secure multi-party computation," *Foundations and Trends® in Privacy and Security*, vol. 2, no. 2-3, pp. 70–246, 2018.

[2] A. C. Yao, "Protocols for secure computations," in *23rd annual symposium on foundations of computer science* (*sfcs 1982*), IEEE, 1982, pp. 160–164.

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

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

주요어: