Secure Two-Party Computation for weak Boneh-Boyen Signature

M. Sečkár[1], and S. Ricci[1]

[1]Department of Telecommunications, Brno University of Technology, Technicka 12, Brno, 616 00, Czech Republic

E-mail: xsecka04@vut.cz, ricci@vut.cz

Abstract—Secure two-party computation allows two entities to securely calculate a common result keeping their private inputs secret. By applying this to the weak Boneh-Boyen signature, a trusted third party is able to sign the user's message (or a secret key) without knowing its content (or value). In this article, we present a C library that implements a two-party computation algorithm for generating a user's secret key that can be used in a group signature scheme. The library provides a structured output ready to be serialized and sent over a network. We also show the computational benchmarks of the implemented algorithms. The computations on the sender's side are relatively fast, which broadens the possibilities of deployment on constrained devices.

Keywords— Secure Multi-Party Computation, Paillier Cryptosystem, Group Signature, C Language, OpenSSL, GMP

1. INTRODUCTION

A digital signature is an algorithm used to verify the integrity of the message and the authenticity of the sender. In addition to these traditional requirements, European Union (EU) regulations and United States (U.S.) strategic plans request new privacy protection features. One way to cover the new demands is to pass to group signatures. For example, an employee (i.e., sender) within a large company needs to sign a message on the behalves of the company without revealing its identity to the external receiver. However, the receiver needs to verify that the sender is an employee of the company and re-identify it in case of malicious behavior. Therefore, we would need that the sender can sign while remaining anonymous (Anonymity) and that the receiver in collaboration with a third trust party can identify malicious senders (Traceability). The signature proposed by Hajny et al. [1] is developed to be efficient on constrained devices and can be turned into a group signature by applying a secure multi-party computation algorithm on the key generation as proposed by Ricci et al. [2].

To the best of our knowledge, we present the first implementation of the secure multi-party computation scheme proposed by Belenkiy et al. [3]. Our implementation has been developed in C language using GNU Multiple Precision Arithmetic Library (GMP) and partly OpenSSL Library. We also show the computational benchmarks of the implemented algorithms.

2. IMPLEMENTATION

In this section, we present our proof-of-concept implementation that is focusing on the secure multi-party computation scheme [3], leaving the proof of knowledge for the second stage of development. The library is located in a public GitHub repository [1]. We also discuss how the parameters should be chosen. The key generation algorithm is run by two entities, Signature Group Manager (SGM) and Senders (Ss). The SGM and each S need to agree on a key that will be used for signing. Belenkiy et al. scheme [3] allows S and SGM to jointly compute the value $\sigma = g^{1/(sk_i+sk_m)}$ of the sender's secret key sk_i and the manager's secret key sk_m without revealing both values. The scheme can be seen as a modified Paillier cryptosystem that provides proof of knowledge of both secrets sk_i and sk_m. Then σ can be used to generate a weak Boneh-Boyen (wBB) signature on a given message. The implementation can be split into two phases: `Setup_SGM` algorithm and `Join` algorithm that are depicted in Algorithms 1 and 2, respectively.

The implementation requires fast computations on large integers. For that reason, C language and specifically GMP library are chosen as a basis for the implementation. The outputs of `Setup_SGM` and

[1]https://github.com/xsecka04/Paillier_NIZKPK

`Join` algorithms are implemented as structures that can be easily serialized and sent over the network. During the `Setup_SGM` phase, the generation of RSA parameters occurs. For this reason `generate_r_from_group`, `generate_r_from_bitlength` and `generate_RSA_SSL` were developed. The RSA parameters are generated using OpenSSL Library, in particular using `BN_generate_prime_ex2` function. Linux's `/dev/urandom` source of pseudo-random numbers is used as a random seed for `mpz_urandomm` function that is located in `generate_r_from_bitlength` and `generate_r_from_group` functions. These functions generate a random number in a given cyclic integer group. Once the RSA parameters are generated, they are encoded in a hexadecimal string from the OpenSSL's `BIGNUM` type and initialized as `mpz_t` integer types to be further used in the GMP functions.

Algorithm 1 `Setup_SGM`(κ)

1: Consider q_{EC} a prime of the right order
2: Generate an RSA-modulus $\mathbf{n}$ of size at least $|2^{3\kappa}q_{EC}^2|$, where $\mathbf{n} = pq$, and $\phi(\mathbf{n}) = (p-1)(q-1)$
3: Consider $\mathbf{h} = \mathbf{n} + 1 \in \mathbb{Z}_{\mathbf{n}^2}$
4: Generate $\mathbf{g}$ of order $\phi(\mathbf{n})$ in $\mathbb{Z}_{\mathbf{n}^2}$
5: Generate another RSA-modulus $\mathfrak{n}$, where p_g, q_g are big primes, $\phi(\mathfrak{n}) = (p_g - 1)(q_g - 1)$
6: Consider $\mathfrak{h} \xleftarrow{\$} \mathbb{Z}_{\mathfrak{n}}$ and $\mathfrak{g} \xleftarrow{\$} \langle \mathfrak{h} \rangle$
7: **return** $par = (\mathbf{h}, \mathbf{n}, \mathbf{g}, \mathfrak{h}, \mathfrak{n}, \mathfrak{g}, q_{EC})$

Algorithm 2 `Join`(par)

1: SGM computes:
2: $r \xleftarrow{\$} \mathbb{Z}_{\phi(\mathbf{n})}, r' \xleftarrow{\$} \mathbb{Z}_{\phi(\mathfrak{n})}$
3: $e_1 = \mathbf{h}^{\mathbf{n}/2+sk_m}\mathbf{g}^r \mod \mathbf{n}^2$,
4: $\mathfrak{c} = \mathfrak{g}^{sk_m}\mathfrak{h}^{r'} \mod \mathfrak{n}$,
5: Sender$_i$ computes:
6: $sk_i, r_1 \xleftarrow{\$} \mathbb{Z}_{q_{EC}}, r_2 \xleftarrow{\$} \{0...|2^{\kappa}q_{EC}|\}$, $\bar{r} \xleftarrow{\$} \{0...|2^{\kappa}\mathfrak{n}|\}$
7: $e_2 = (e_1/\mathbf{h}^{\mathbf{n}/2})^{r_1}\mathbf{h}^{\mathbf{n}/2+sk_ir_1+r_2q_{EC}}\mathbf{g}^{\bar{r}} \mod \mathbf{n}^2$
8: $\mathfrak{c}' = \mathfrak{g}^{sk_i}\mathfrak{h}^{\bar{r}} \mod \mathfrak{n}$,
9: SGM computes:
10: $x = Dec(e_2) - \mathbf{n}/2$

GMP contains the multiple precision integer `mpz_t` data type used for the storage and computations. Multiple precision (or Arbitrary precision) means that the size can grow dynamically and is not restricted as other generic fixed precision data types. For the variable to be properly used, it needs not only to be declared (`mpz_t variable`), but also initialized (`mpz_init(variable)`) and cleared after use (`mpz_clear( variable)`). GMP library provides several integer operation functions such as arithmetic (e.g., `mpz_add()`), exponentiation (e.g., `mpz_powm()`), modular operations (e.g., `mpz_invert()`), and pseudo-random number generators (e.g., `mpz_urandomm()`).

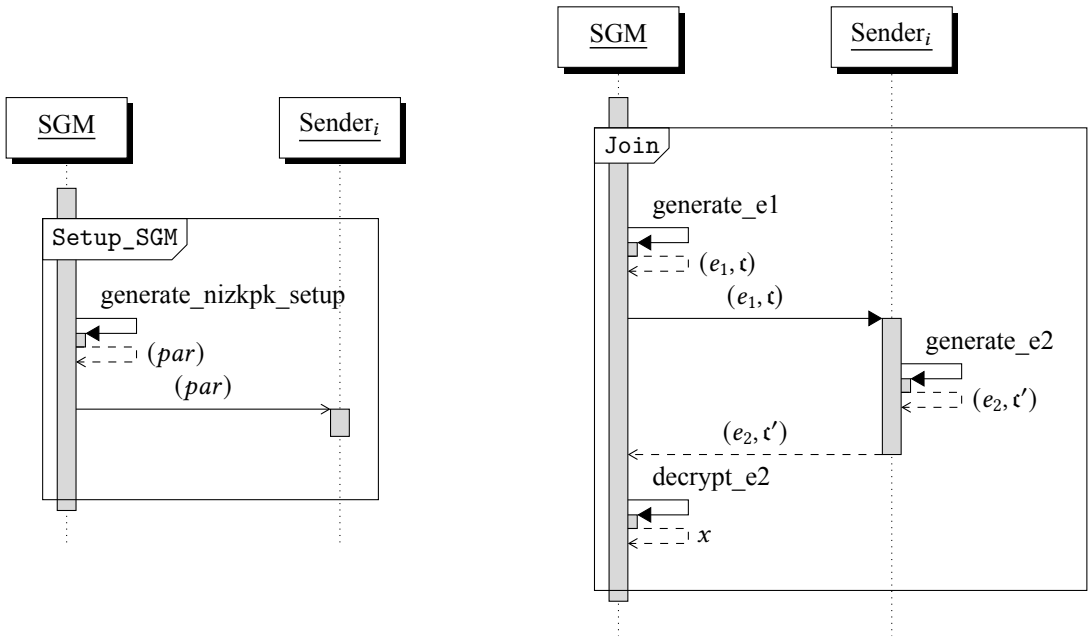

Figure 1: From left to right, the sequence diagrams of `Setup_SGM` and `Join` algorithms

Figure 1 depicts the sequence diagrams of `Setup_SGM` and `Join` algorithms, respectively. The parameters

sent between SGM and Sender$_i$ are serialized into JSON objects as hexadecimal strings using `JSON_serialize_Setup_par`, `JSON_serialize_e1` and `JSON_serialize_e2` functions. On the other end, the objects are being deserialized using their respective deserialization functions, i.e. `JSON_deserialize_Setup_par`. Note that the use of these functions is optional and the end user can choose different formats for the serialization.

Algorithm 2 requires the homomorphic computation of σ. As suggested by Belenkiy et al. [3], we consider the Paillier cryptosystem. In particular, in order to securely compute σ, $\mathbf{h}$ has to be taken equal to $\mathbf{n}+1$ and $\lambda = \phi(\mathbf{n})$. The decryption needs also to be adjusted accordingly. These modifications are directly suggested by Paillier [4].

2.1. Parameters Setup

The scheme requires two RSA groups, one for the homomorphic calculations and the other for the commitment parameters. Parameters $\mathfrak{h}, \mathfrak{n}, \mathfrak{g}$ can be utilized in the non-interactive zero-knowledge proofs of knowledge, alongside the challenges e_1, e_2 and commitments $\mathfrak{c}$ and $\mathfrak{c}'$. We refer to [3] for more details.

Algorithm 1 shows how the parameters should be chosen. There are several steps needed to be specified in order to implement the depicted `Setup_SGM` scheme. Note that the RSA-modulus $\mathbf{n}$ has bitlength of at least $|2^{3\kappa}q_{EC}^2|$. Following the NIST security standard, we consider q_{EC} a prime of 256 bitlength and a security parameter κ equal to 1350. This brings the RSA-modulus $\mathbf{n}$ to be large enough to make the encryption and decryption secure and follow the NIST standards.

In Step 4 of Algorithm 1, we need to generate $\mathbf{g}$ of order $\phi(\mathbf{n})$ in $\mathbb{Z}_{\mathbf{n}^2}$. Finding an element of order $\phi(\mathbf{n})$ in a big composite group is not an easy task. The traditional approach for finding a generator on $\mathbb{Z}_{\mathbf{n}^2}$ requires the selection of a random number h in the group and the check if h is of the right order. It is important to notice that a composite group has not guaranteed to possess generators and that the search requires exponentiations in a big modulus that are computationally demanding. We solved this issue by considering how Paillier cryptosystem [4] generates the noise in the encryption. Note that the implemented algorithm is a variant of Paillier scheme. Therefore, we generated a random number $k \xleftarrow{\$} \mathbb{Z}_{\mathbf{n}^2}$ and we computed $\mathbf{g} \leftarrow k^{\mathbf{n}} \bmod \mathbf{n}^2$. In this way, $\mathbf{g}^{\phi(\mathbf{n})} \equiv 1 \bmod \mathbf{n}^2$, i.e., has the right order for the computations.

In case of the second RSA-modulus (Step 5), we also considered $\mathfrak{n}$ of size at least $|2^{3\kappa}q_{EC}^2|$ and we generate $\mathfrak{h} \xleftarrow{\$} \mathbb{Z}_{\mathfrak{n}}$. Moreover, $\mathfrak{g}$ has to be taken from the subgroup generated by $\mathfrak{h}$ (please see Step 6 in Algorithm 1) and, therefore, we generated a random $t \xleftarrow{\$} \mathbb{Z}_{\phi(\mathfrak{n})}$ and we computed $\mathfrak{g} = \mathfrak{h}^t$. Once the parameters are set, Algorithm 2 can be applied to compute $\sigma = g^{1/(sk_i+sk_m)}$ without revealing sk_i to SGM and sk_m to S. If the assertion $x \equiv ((sk_m + sk_i)r_1 \mod \mathbf{n}) \mod q_{EC}$ holds, the exchange has been successful and the parties are ready for a generation of the signature and verification using the wBB signature. Moreover, $\mathbf{n}/2$ calculation is implemented as $\lfloor \mathbf{n}/2 \rfloor$ and $(e_1/\mathbf{h}^{\mathbf{n}/2})$ as $e_1(\mathbf{h}^{\mathbf{n}/2})^{-1} \mod \mathbf{n}^2$, where x^{-1} denotes a modular multiplicative inverse and $\lfloor ... \rfloor$ denotes the floor function. The former change was made due to the fact that 2 has no multiplicative inverse in $\phi(\mathbf{n}^2)$ and, therefore, $\mathbf{n}/2 \mod \phi(\mathbf{n}^2)$ cannot be computed. Table I shows the chosen set of parameters and their respective sizes. Our choices follow the NIST security standard.

Table I: Input parameters for `Setup_SGM` algorithm

Parameter	Size [Bits]	Value
κ (Security Parameter)	32	1350
q_{EC} (Order of the Elliptic curve)	256	0x2523648240000001ba344d8000000007ff9f800000000010a10000000000000d[1]
$\mathbf{n}, \mathfrak{n}$ (RSA Moduli)	4572	random
$\mathbf{g}$	9139	random
$\mathbf{h}$	4572	$\mathbf{n}+1$
$\mathfrak{h}, \mathfrak{g}$	4572	random

[1]The value is given in the hexadecimal representation.

3. EXPERIMENTAL RESULTS

The tests were conducted on two devices: AMD Ryzen 9 5900X CPU with 64GB RAM in a Docker environment built on Debian GNU/Linux 10 (buster) with 5.10.16.3-microsoft-standard-WSL2 kernel and on a Raspberry Pi 4 Model B - 4GB RAM built on Raspberry Pi OS. For the purpose of testing the computational time, `time.h` from the C standard library was utilized as well as a simple script looping over the computations and storing them into arrays of structures. Table II depicts elapsed CPU time from the tests with input parameters shown in Table I, which also depicts the bit-lengths of the RSA moduli. The depicted results in Table II were calculated as a mean over 10 runs. Note that the network overhead is not accounted for in these calculations.

Table II: Performance benchmarks

Environment	Number of senders	Elapsed Time of `Setup_SGM` [ms]	Elapsed Time of `Join` (SGM) [ms]	Elapsed Time of `Join` (Sender) [ms]
Docker	1	3330.96	211.13	194.37
	50	2810.68	10341.12	9662.89
	100	3128.50	20655.89	19286.74
Raspberry Pi	1	24489.47	2915.55	2721.11
	50	20472.66	145558.53	135836.86
	100	20974.48	290718.96	271411.49

As shown in Table II, the most time-consuming computations are done on the SGM side and the computing requirements on the sender's side are relatively negligible. The table shows the time complexity independence of `Setup_SGM` algorithm and the linear dependence of `Join` algorithm on the number of senders. If we consider the elapsed time of `Setup_SGM` algorithm, the variation between 1 and 100 senders is due to the fact that some of the parameter generators do not take a constant time to compute. Moreover, the serialization of the parameters accounts for around 6 ms in each serialization/deserialization pair. This provides the possibility of deploying the algorithm on microprocessors for a wider array of use cases. Due to the fact that `Join` algorithm has to be run only once, 2.7 seconds can still be considered feasible on constrained devices.

4. CONCLUSION

To the best of our knowledge, we present the first implementation of the secure two-party computation scheme proposed by Belenkiy et al. [3]. To do so, we developed a library that provides a structured output ready to be serialized and sent over a network. Moreover, we presented the performance benchmarks without networking overhead and showed that the computing beared on the sender is minimal.

As future work, we will continue with the design of the PKs and their implementation. Moreover, we will show how the verification phase can be accomplished on elliptic curves using bilinear pairings as well as modeling the network communication.

REFERENCES

[1] J. Hajny, P. Dzurenda, L. Malina, and S. Ricci, "Anonymous Data Collection Scheme from Short Group Signatures", *Proceedings of the 15th International Joint Conference on e-Business and Telecommunications*, pp. 200-209, Jul. 2018.

[2] S. Ricci, P. Dzurenda, J. Hajný, and L. Malina, "Privacy-Enhancing Group Signcryption Scheme", *IEEE Access*, vol. 9, no. 10, pp. 136529-136551, 2021.

[3] M. Belenkiy, J. Camenisch, M. Chase, M. Kohlweiss, A. Lysyanskaya, and H. Shacham, "Randomizable Proofs and Delegatable Anonymous Credentials", in *Advances in Cryptology - CRYPTO 2009*, vol. 5677, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 108-125.

[4] P. Paillier, "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes", in *Advances in Cryptology - EUROCRYPT -99*, 1999, vol. 1592, pp. 223-238.