

The return of Eratosthenes: Secure Generation of RSA Moduli using Distributed Sieving

Cyprien Delpech de Saint Guilhem
imec-COSIC, KU Leuven
Belgium
cdsg@esat.kuleuven.be

Dragos Rotaru
imec-COSIC, KU Leuven
Belgium
Cape Privacy
New York, United States
dragos@capeprivacy.com

Eleftheria Makri
imec-COSIC, KU Leuven
Belgium

ABRR, Saxion University of Applied Sciences
The Netherlands
emakri@esat.kuleuven.be

Titouan Tanguy
imec-COSIC, KU Leuven
Belgium
ttanguy@esat.kuleuven.be

ABSTRACT

Secure multiparty generation of an RSA biprime is a challenging task, which increasingly receives attention, due to the numerous privacy-preserving applications that require it. In this work, we construct a new protocol for the RSA biprime generation task, secure against a malicious adversary, who can corrupt any subset of protocol participants. Our protocol is designed with generic multiparty computation (MPC), making it both platform-independent and allowing for weaker security models to be assumed (e.g., honest majority), should the application scenario require it. By carefully “postponing” the check of possible inconsistencies in the shares provided by malicious adversaries, we achieve noteworthy efficiency improvements. Concretely, we are able to produce additive sharings of the prime candidates, from multiplicative sharings via a semi-honest multiplication, without degrading the overall (active) security of our protocol. This is the core of our sieving technique, increasing the probability of our protocol sampling a biprime. Similarly, we perform the first biprimality test, requiring several repetitions, without checking input share consistency, and perform the more costly consistency check only in case of success of the Jacobi symbol based biprimality test. Moreover, we propose a protocol to convert an additive sharing over a ring, into an additive sharing over the integers. Besides being a necessary sub-protocol for the RSA biprime generation, this conversion protocol is of independent interest. The cost analysis of our protocol demonstrated that our approach improves the current state-of-the-art (Chen et al.—Crypto 2020), in terms of communication efficiency. Concretely, for the two-party case with malicious security, and primes of 2048 bits, our protocol improves communication by a factor of ~ 37 .

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

© 2021 Association for Computing Machinery.

ACM ISBN 978-1-4503-8454-4/21/11... \$15.00

<https://doi.org/10.1145/3460120.3484754>

CCS CONCEPTS

• Theory of computation → Cryptographic protocols.

KEYWORDS

RSA modulus, Multiparty Computation, Active Security

ACM Reference Format:

Cyprien Delpech de Saint Guilhem, Eleftheria Makri, Dragos Rotaru, & Titouan Tanguy. 2021. The return of Eratosthenes: Secure Generation of RSA Moduli using Distributed Sieving. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*, November 15–19, 2021, Virtual Event, Republic of Korea. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3460120.3484754>

1 INTRODUCTION

An *RSA modulus*, also known as a *biprime*, and usually denoted by the variable N , refers to a number which is the product of two prime numbers, usually denoted by p and q ; thus, $N = p \cdot q$. The RSA modulus is a crucial component of the first public key encryption scheme, the RSA scheme [33], as well as many other public key encryption schemes that followed it. The security of the RSA cryptosystem is based on the hardness of factoring, and as such N is part of the public key of the RSA scheme [33], while its factors p and q determine the secret key. Specifically, the security of the cryptosystem is determined by the bit-length of the biprime, and therefore efficient methods to generate (large) biprimes have been of interest since RSA was devised.

Initially, the generation of the parameters of a public key cryptosystem (including the biprime generation) was assigned to a trusted third party. However, there are applications where no single party can be entrusted with such a task, which gave rise to the study of distributed biprime generation. The problem of secure distributed RSA modulus generation is being studied since 1997, when the seminal work of Boneh and Franklin [5] appeared. After the initial interest in the problem, in the years around the work of Boneh and Franklin, with literature attempting to improve the efficiency, or security aspects of the original work, the subject ceased being studied for about a decade. Then, again, during the last decade the interest on secure

distributed RSA modulus generation is increasing. This is due to the sheer number of recent applications, requiring distributed RSA modulus generation.

Traditionally, the study of distributed RSA modulus generation has found numerous applications in threshold cryptography [13–15, 19, 32]. Nowadays, blockchain applications requiring permissionless consensus, also need to deploy the techniques of threshold cryptography, which explains the recently revived interest in RSA modulus generation. A concrete example of applications for the RSA modulus generation in the context of decentralized systems and consensus protocols is that of Verifiable Delay Functions (VDFs) [4, 30, 39]. Threshold cryptography requiring the distributed generation of an RSA modulus is now expanding beyond academia, as companies and foundations (e.g., Unbound, the VDF Alliance, the Ethereum Foundation, Ligerio) are providing services based on top of these technologies to the public.

1.1 Related Work

The study of secure multiparty RSA modulus generation was initiated by Boneh and Franklin [5]. Boneh and Franklin [5] devised a biprimality test to perform the distributed RSA modulus generation, instead of individually testing the primality of the two prime factors of N . Given that N is the public output of their protocol, this granted them an efficiency advantage, since their expensive multiparty computations can be computed modulo the public N this way. On the other hand, the biprimality testing approach, requires that two prime numbers are *simultaneously* sampled, which leads to an increased number of iteratively invoking the subroutines of the protocol, since efficient primality testing is probabilistic in nature. Trial division, applied individually on each of the prime candidates, somewhat relaxes the abovementioned performance penalty. The blueprint of Boneh and Franklin [5], which is also adopted by most of the follow-up protocols in the literature, consists of three main steps: (1) pick prime candidates (via trial division); (2) securely multiply candidates; (3) biprimality testing (followed by the RSA key generation step, whenever key generation is actually needed).

The Boneh-Franklin protocol was implemented, together with some newly introduced optimizations by Malkin et al. [29]. Malkin et al. [29] first deploy a (simpler) Fermat test to check biprimality, which with low probability introduces false positives. If this simpler test passes, then they deploy the Boneh-Franklin biprimality test, to eliminate any potential false positives. The most important optimization proposed by Malkin et al. [29] is a distributed sieving technique, which results in a $10\times$ improvement in running time for the generation of a 1024-bit biprime. The distributed sieving ensures that the candidate primes p and q are not divisible by the first small primes, up to a predetermined bound. This is done by each party randomly selecting multiplicative shares, which are coprime to the predetermined bound (and therefore their product is also coprime to the bound), and then transforming these to additive shares to proceed with the rest of the protocol.

Frankel et al. [16] were the first to propose a distributed RSA key generation in the malicious security, honest majority setting. One of the main tools, devised and used for the RSA key generation by Frankel et al. [16], is an unconditionally secure multiplication protocol over the integers. To generate the RSA modulus Frankel et

al. [16] deploy a maliciously secure version of the Boneh-Franklin biprimality test, and then show how to produce the actual RSA keys: a more efficient version for small keys, and a less efficient one for larger keys. Although their tailored protocol is more efficient than a solution merely deploying a passive to active security compiler, it still remains inefficient.

Limited to the two-party case, Poupard and Stern [31] propose a maliciously secure protocol for the RSA modulus generation, based on OT. Although their protocol is less efficient than the Boneh-Franklin one, it is secure in a more stringent security model, and it can serve two parties (instead of three that are required by the Boneh-Franklin protocol), in application scenarios where this is needed. However, the protocol suffers from a leakage of information in the presence of a malicious party, who can learn up to $\sum_{p \in \mathcal{P}} \log(p)/p$ bits of the prime factor p , with \mathcal{P} being the set of tested primes. Another OT-based, two-party protocol for the RSA key generation (and the RSA modulus generation) was proposed by Gilboa [20]. Unlike Poupard and Stern's solution, Gilboa's protocol offers only semi-honest security, but it is more efficient. The well-known OT-based multiplication protocol of Gilboa, which by now is a classic, and adaptations thereof are frequently used in the construction of secure multiparty computation protocols, is also the basis of the RSA key generation protocol that they devised.

Algesheimer et al. [1] perform a distributed primality test, unlike the protocol of Boneh and Franklin [5] that was based on biprimality testing. Specifically, they show how to perform a distributed version of the Miller-Rabin primality test, and do proceed with the multiparty computations modulo a secret prime, that the previous work avoided. To achieve this, they deploy three types of secret sharing schemes, and show how to convert shares from one to another. Moreover, their constructions allow the generation of an RSA modulus, whose prime factors are actually *safe* primes. Both the work of Algesheimer et al. [1] and the work of Boneh and Franklin [5] are proven secure in the semi-honest, honest majority security model, and require minimally three parties.

Damgård and Mikkelsen [12] were the first to efficiently achieve malicious security for the task of distributed RSA modulus generation, though in the honest majority setting. In fact, the first to adjust the protocol of Boneh and Franklin to the active security model were Frankel et al. [16], but malicious security was achieved at the cost of protocol efficiency. The work of Damgård and Mikkelsen [12] can be seen as a hybrid of two of its predecessors [1, 5], trying to combine the most efficient aspects of both of these approaches. Concretely, by using replicated secret sharing they can do both multiplications, and the modular reductions suggested by Algesheimer et al. [1], which require the protocol to work over the integers (instead of a field), without having to convert between different secret sharing schemes. The downside of the Damgård and Mikkelsen's [12] protocol is that it does not scale well in the number of parties, and it is not straightforward to extend to more than three parties.

Hazay et al. [22, 23] deploy partially homomorphic encryption to complete the two first steps of the Boneh and Franklin blueprint, and proceed to step 3 with biprimality testing. They achieve the first general n -party protocol (i.e., for any $n \geq 2$) in the active security model, and the dishonest majority setting. Active security in this setting is achieved by deploying tailored zero-knowledge proofs. Hazay et al. [22, 23] build upon Gilboa's technique [20] achieving

two-party actively secure RSA modulus generation à la Boneh and Franklin, but they are the first to adapt also the trial division step of the protocol, achieving therefore a significant efficiency boost. For the multiplication step that follows to compute the candidate biprime, Hazay et al. [22, 23] carefully use a combination of the Paillier and ElGamal encryption schemes.

In the two party setting, Frederiksen et al. [17] propose an OT-based, maliciously secure protocol for the distributed RSA modulus generation, which is more efficient than previous work. The efficiency improvement is due to one single compact zero-knowledge argument of correct behavior at the end of the protocol, instead of the numerous (one per message) tailored zero-knowledge proofs required in the protocol of Hazay et al. [22, 23]. The maliciously secure protocol of Frederiksen et al. [17] is provably secure, and concretely efficient, as shown by the authors with an implementation. However, it suffers from some information leakage at the trial division step. This leakage is formalized in the functionality, and taken into account in the security proof, and it is argued to be justified both in theory and in practice, as leakage of a few bits of the prime factors should not be able to break the RSA assumption, and therefore also not the security of the protocol. Nevertheless, in the malicious case, this leakage may lead to selective failure attacks, which also impacts the efficiency of the protocol.

Recently, Chen et al. [7] successfully overcame the limitations of the proposal of Frederiksen et al. [17]. Benefiting from the efficiency advantages that a CRT number representation allows, Chen et al. [7] devise a maliciously secure n -party protocol, tolerating $n-1$ (active) corruptions, while avoiding both the deployment of expensive cryptographic primitives, and the information leakage incurred by the protocol of Frederiksen et al. [17]. Leveraging the CRT representation, Chen et al. [7] not only gain efficiency from the linearity of the smaller in bit-size computations that can be performed locally, but they also *constructively* sample their primes such that they are not divisible by the small primes used for the CRT representation. This way they significantly increase the probability of hitting a prime, and therefore also increase the overall protocol efficiency. Their distributed RSA modulus generation protocol follows the general Boneh and Franklin paradigm.

A large scale implementation of the distributed RSA modulus generation, and improvement of the previous work of Chen et al. [7], named Diogenes [8], is the most recent result in the area. From a scalability point of view, Diogenes [8] is the first MPC implementation for a non-trivial task that scales to thousands of parties. To achieve efficiency and scalability, Diogenes [8] deploys the so-called *coordinator model*, which is a setting consisting of a powerful, honest-but-curious coordinator, and thousands of relatively lightweight computation parties. The RSA modulus generation protocol proposed in Diogenes [8] is secure against $n-1$ (out of the n) malicious parties. Building upon the work of Chen et al. [7], Diogenes [8] also deploys the CRT representation, and constructive sampling techniques. To avoid the communication cost that pairwise messages incur, and to exploit the potential of packing and SIMD to the fullest, Diogenes [8] is based on a Ring LWE additively homomorphic encryption scheme (AHE), where the semi-honest coordinator is tasked to perform all the homomorphic additions necessary, as well as relay messages. Malicious security for such a large scale application, is achieved by a composition of zero-knowledge techniques, the certification of

Protocol	Security	DM	#Parties	Test	NL
[5]	Passive	×	$n \geq 3$	biprimality	✓
[16]	Active	×	$n \geq 3$	biprimality	✓
[31]	Active	✓	$n = 2$	biprimality	×
[20]	Passive	✓	$n = 2$	biprimality	✓
[1]	Passive	×	$n \geq 3$	primality	✓
[12]	Active	×	$n = 3^*$	primality	✓
[22]	Active	✓	$n \geq 2$	biprimality	✓
[23]					
[17]	Active	✓	$n = 2$	biprimality	×
[7]	Active	✓	$n \geq 2$	biprimality	✓
[8]	Active**	✓	$n \geq 2$	biprimality	✓
Ours	Active	✓	$n \geq 2$	biprimality	✓

Table 1: Comparison of the related work. DM stands for Dishonest Majority, and NL stands for No Leakage.

which is aggregated and verified only once, at the end of the protocol, and only for the successful protocol iteration. In Table 1 the main functionality and security features of our work and the related works are summarized.

1.2 Our Contribution

In this work we show how to securely generate an RSA biprime in the standard multiparty setting, where all parties contribute equally to the computation. We assume a static active adversary who can corrupt up to $n-1$ (out of the total n) parties, but remark that our proposal works with generic MPC, allowing the deployment of different security models, based on the needs of the application at hand. This makes our protocol MPC-platform-independent, as it can be realized with any MPC technology that is based on linear secret sharing techniques. For example, Shamir's secret sharing [37] can be deployed, if our goal is to produce the RSA moduli in the honest majority setting; or (a variant of) the replicated secret sharing scheme of Araki et al. [2, 18], should high throughput be the main goal of the MPC implementation.

Following the paradigm of recent work [7], we design a constructive distributed sampling sub-protocol that increases the probability of our overall protocol generating a biprime. Crucially, we achieve this constructive sampling having the parties first sample multiplicative sharings of a certain form, and then transforming them into additive sharings, by computing their product in a semi-honest fashion. This does not degrade the security of the RSA generation protocol, because subsequently we reveal the public biprime N (i.e., the product of the sampled candidate primes p and q). An adversary who succeeds in introducing an additive error in the sharings of p or q that is consistent with the error in their product N , should effectively factor N , which is hard by the original assumption for an RSA biprime. This semi-honest multiplication presents itself as a major bulk of the protocol's cost, so the savings from performing it semi-honestly are substantial.

*The protocol can be non-trivially extended to support more than 3 parties, but efficiency does not scale.

**Diogenes works in the semi-honest coordinator model, and active security is only guaranteed for the non-coordinating parties.

Another important technique we deploy is to run the biprimality test in terms of checking the Jacobi symbol (which identifies most of the biprimes successfully) without checking the consistency of the input shares in the test. Note that the Jacobi test has to be repeated once for each candidate and σ times for a candidate that passes the first iteration of the test, for σ a statistical security parameter. This means that any cost savings in this part of the protocol impact significantly the overall efficiency. The more computationally and communication intensive consistency checks are only performed on the candidate for which all repetitions of the Jacobi test have succeeded.

To perform the consistency check that follows the Jacobi symbol test, our protocol requires to convert a bounded additive sharing from its CRT representation to a single additive sharing over the integers. This is to match the computations performed in the exponent over the integers for the completion of the Jacobi test. We design a protocol that performs the aforementioned conversion, and we remark that in addition to being necessary for our RSA modulus generation, this protocol is of independent interest. For example, one of the PRF constructions in Grassi et al. [21], requires the MPC preprocessing field to be compatible with an elliptic curve group \mathbb{G} . Our exponentiation protocol, with public output and secret exponent, would make their preprocessing field compatible with the latest SHE techniques [3, 28], since those require special primes, which might be incompatible with elliptic curve groups. Our work might also improve the preprocessing efficiency in other works, which need to compute g^x in public, where x is secret shared [10, 24, 38], but we leave this for future investigation.

Lastly, we analyze our protocol, and set concrete parameters to compute the communication cost it incurs. We also show how the communication cost of our protocol scales in the number of parties, and for different parameter sets. With conservative estimations, and a statistical security parameter set to $\sigma = 80$, our protocol outperforms the current state-of-the-art [7] in all but one settings: the semi-honest security with 16 parties setting. For malicious security, and primes of 2048 bits, our protocol improves the previous work by over 30 times, both in the two-party, and in the 16-party case.

To summarize, our main contributions are as follows:

- (1) Our RSA modulus generation protocol works for generic MPC, being able to leverage any MPC technology based on linear secret sharing.
- (2) We constructively sample candidate primes, transforming multiplicative sharings to additive sharings, by computing their product in a semi-honest fashion, which is checked for maliciously inserted additive errors later in the protocol, resulting in the protocol's cost reduction.
- (3) The first biprimality check, implemented by means of checking the Jacobi symbol, is costly and repeated σ times in our protocol. We show how to postpone the even costlier consistency check on the shares contributed to the Jacobi test, in order to again gain efficiency.
- (4) We design a protocol to convert an additive sharing over a ring to an additive sharing over the integers, which is of independent interest.
- (5) We demonstrate that our protocol improves the communication cost over the current state-of-the-art [7].

1.3 Technical Overview

Our main protocol, Π_{RSAGen} , works in five distinct phases: (1) the sampling phase, aiming at generating two prime numbers p and q , secret shared among the protocol participants; (2) the combine phase, computing the product N of the previously sampled candidate primes, which is securely computed and then revealed to all parties; (3) the Jacobi test, checking whether the computed N is a biprime; (4) the consistency check, ensuring input consistency in the presence of malicious adversaries, should the Jacobi test indicate a candidate biprime; and (5) the GCD test, which checks again whether N is a biprime, to ensure that the protocol did not accept a false positive that the Jacobi test may not catch.

Our Sampling phase first deploys a technique similar to the one introduced by Malkin et al. [29], which they term *distributed sieving*. Distributed sieving entails each party sampling a multiplicative share for each of the two primes p and q , then performing a (semi-honest) multiplication on these shares, and then re-share them to transform them into additive shares. With the distributed sieving we increase the probability of sampling primes p and q . Similarly to recent related work [7, 8], we leverage the Chinese Remainder Theorem (CRT) to further increase the efficiency of our protocol. To this end, we show how to extend the standard actively secure MPC functionality to work on separate MPC engines: one for each of the CRT components we consider. We call this functionality $\mathcal{F}_{\text{MPC-CRT}}$.

In the Combine phase of Π_{RSAGen} , based on the aforementioned $\mathcal{F}_{\text{MPC-CRT}}$ functionality, we perform an actively secure multiplication between the two sampled primes, we reveal the result to all parties, and check whether the product falls within the predetermined bounds, and whether it is coprime to a value M_{sample} , which is the product of $\ell(\lambda)$ primes. Should both of these checks pass, the combine phase is completed and we proceed to the Jacobi test on the generated candidate biprime.

The Jacobi test aims at establishing whether the product N is a biprime. Although this test introduces no false negatives, it has a probability of $1/2$ of introducing a false positive (i.e., accepting a non-biprime). To increase the probability of N being a biprime to $2^{-\sigma}$ (before proceeding to the ultimate GCD test) we repeat the Jacobi test σ times. The core of the Jacobi test lies in a secure exponentiation protocol, with public output, where the computations in the exponent are performed over the integers. We deploy the exponentiation protocol proposed by Grassi et al. [21] to compute the desired Jacobi symbol. If the Jacobi symbol is ± 1 , we proceed to the next phase, which is the Consistency Check.

The consistency check ensures that the protocol will abort, in the presence of active adversaries who have input inconsistent shares of the candidate primes. To achieve this, we carefully mask the exponent of the Jacobi test, with bounded randomness (for which we have devised a specialized protocol, Π_{RandZk}) so that all computations are performed over the integers without wrap around. Then, the masked value itself needs to be an additive sharing over the integers. To this end, we have devised a protocol to convert an additive sharing over a ring, into an additive sharing over the integers, named Π_{ConvInt} . By ensuring that indeed no computation wrapped around, we check an equivalent relationship for the exponentiation performed for the

Algorithm CRTrec($(x_{p_1}, \dots, x_{p_\ell}), (p_1, \dots, p_\ell)$)

- (1) Compute $N = \prod_{i=1}^{\ell} p_i$.
- (2) For all $i \in \{1, \dots, \ell\}$ compute $N_i = N/p_i$ and find M_i satisfying $N_i \cdot M_i = 1 \pmod{p_i}$.
- (3) Compute $x = \sum_{i=1}^{\ell} x_{p_i} N_i M_i \pmod{N}$.

Figure 1: CRT Reconstruction Algorithm.

Jacobi symbol computation, which serves as a proof of input consistency of the shares contributed by each party to the Jacobi test. This ensures security against malicious adversaries.

The last phase of our protocol aims at eliminating any false positives that are not filtered out by the Jacobi test. Concretely, in the GCD phase we wish to verify that $\gcd(N, (p+q+1)) = 1$. This phase requires again the generation of bounded randomness, for which we deploy the same protocol we devised for the Jacobi test, as well as a careful selection of the bounds, and number of necessary CRT components, ensuring that no wrap around happens during the secure computations. Note that according to the original work of Boneh and Franklin [5], the latter test introduces a false negative, in the particular case of $N = p \cdot q$, with p, q primes, and $q = 1 \pmod{p}$.

2 PRELIMINARIES

2.1 Chinese Remainder Theorem - CRT

Following the blueprint of the two most recent works in distributed RSA modulus generation [7, 8], we deploy in our work the Chinese Remainder Theorem to increase the efficiency of our protocol. We recall here the Chinese Remainder Theorem [25].

THEOREM 2.1. *Let $N = pq$ where p and q are relatively prime. Then $\mathbb{Z}_N \simeq \mathbb{Z}_p \times \mathbb{Z}_q$ and $\mathbb{Z}_N^* \simeq \mathbb{Z}_p^* \times \mathbb{Z}_q^*$. Moreover, let f be the function mapping elements $x \in \{0, \dots, N-1\}$ to pairs (x_p, x_q) with $x_p \in \{0, \dots, p-1\}$ and $x_q \in \{0, \dots, q-1\}$ defined by $f(x) = ([x \pmod{p}], [x \pmod{q}])$. Then f is an isomorphism from \mathbb{Z}_N to $\mathbb{Z}_p \times \mathbb{Z}_q$, as well as an isomorphism from \mathbb{Z}_N^* to $\mathbb{Z}_p^* \times \mathbb{Z}_q^*$.*

The CRT generalizes to any vector of pairwise relatively primes p_1, p_2, \dots, p_ℓ , whose product is $N = \prod_{i=1}^{\ell} p_i$. Then the function f mapping elements $x \in \{0, \dots, N-1\}$ to tuples $(x_{p_1}, \dots, x_{p_\ell})$ with $x_{p_j} \in \{0, \dots, p_j-1\}$, is an isomorphism from \mathbb{Z}_N to $\mathbb{Z}_{p_1} \times \dots \times \mathbb{Z}_{p_\ell}$ and from \mathbb{Z}_N^* to $\mathbb{Z}_{p_1}^* \times \dots \times \mathbb{Z}_{p_\ell}^*$. We refer to the tuples $(x_{p_1}, \dots, x_{p_\ell})$ as the CRT representation of x .

To convert an element from its CRT representation to its representation mod N , we deploy the so-called CRT Reconstruction algorithm, which is presented in Fig. 1.

2.2 Notation

We define $M_{\text{sample}} = 3 \cdot 5 \cdots p_b$ to be the product of the first b primes (excluding 2). This is the space from which we sample the first multiplicative sharings of the candidate primes p and q in our protocol. Further, we define $M_\ell = p_1 \cdot p_2 \cdots p_\ell$ to be the product of ℓ distinct primes of size 128 bits each. To achieve efficient arithmetic over M_ℓ we use ℓ distinct MPC engines, each of which operates over p_i . At different stages of our protocols we work either with these distinct MPC engines, or we perform the CRT reconstruction of the variables we work with over an MPC engine M_ℓ .

To compress and simplify notation throughout this paper, we denote by (x, ℓ) the CRT representation of $x \pmod{M_\ell}$, that is all ℓ CRT components $(x \pmod{p_1}, \dots, x \pmod{p_\ell})$. The local operation of CRT reconstruction of $x \pmod{M_\ell}$ from its CRT representation is denoted as CRTrec(x, ℓ).

We use square brackets to denote additively secret shared values, e.g., the shared version of x is denoted by $[x]$. We use double square brackets for the *authenticated* secret shared values, e.g., the authenticated shared version of x is denoted by $\llbracket x \rrbracket$. When the sharings are over the CRT representation with ℓ CRT components, we denote the sharings as $[x, \ell]$, and $\llbracket x, \ell \rrbracket$, respectively, and assume ℓ MPC engines operating in parallel, one for each CRT component.

3 PROTOCOL INGREDIENTS

Our main protocol for the biprime generation depends on several functionalities. In this section we present all functionalities that are necessary for the realization of the final $\mathcal{F}_{\text{RSAGen}}$ functionality, and elaborate on the non-standard ones. We begin the description of the ingredients that comprise our final protocol with a roadmap explaining the dependencies between the functionalities required to realize $\mathcal{F}_{\text{RSAGen}}$.

3.1 Roadmap

In Fig. 2 we demonstrate the functionality dependencies for the RSA modulus generation. We denote functionalities with circles, and protocols with rectangles. On the dependency vectors 'H' stands for hybrid (as in which hybrid model do we assume for the protocol), and 'R' stands for realizing, and leads to the functionality that the origin protocol realizes. In this section we show how to reach the root of the depicted tree, namely the Π_{RSAGen} protocol, which in turn realizes the $\mathcal{F}_{\text{RSAGen}}$ functionality.

The first functionality that our protocol makes use of, is the $\mathcal{F}_{\text{ABWWithErrors}}$. This is used in the sampling phase of Π_{RSAGen} , where we resort to a semi-honest multiplication protocol to compute the additive shares of the two primes contributed by each party, from their multiplicative shares. This is realized by the $\Pi_{\text{ABWWithErrors}}$ protocol, which in turn is constructed in the $\mathcal{F}_{\text{ABWWithErrors-Prep}}$ hybrid model (realized by $\Pi_{\text{ABWWithErrors-Prep}}$). The reader can think of this functionality as the standard MPC arithmetic black-box, secure against passive adversaries. The preprocessing phase of the arithmetic black-box produces unauthenticated input tuples, and multiplication triples. We elaborate on the workings of $\Pi_{\text{ABWWithErrors}}$ in Section 3.2.

Then, the rest of the sampling phase, as well as the combining phase of Π_{RSAGen} uses the $\mathcal{F}_{\text{MPC-CRT}}$ functionality, realized by the $\Pi_{\text{MPC-CRT}}$ protocol, which is in turn designed in the standard \mathcal{F}_{MPC} hybrid model. For completeness, we present the \mathcal{F}_{MPC} functionality in Fig. 13, Appendix B. In Section 3.3 we show how to generalize the standard actively secure MPC functionality to support parallel MPC engines operating over sharings of the CRT representation of the inputs, designing therefore the $\mathcal{F}_{\text{MPC-CRT}}$ functionality.

The Jacobi test phase of Π_{RSAGen} makes use of the standard broadcast, and randomness sampling functionalities, which are presented for completeness in Appendix B, Fig. 14, and Fig. 15, respectively. The consistency check that follows the Jacobi test of Π_{RSAGen} requires two additional functionalities. To support these two additional

functionalities, we augment the $\mathcal{F}_{\text{MPC-CRT}}$ functionality with two additional commands, and integrate them into the $\Pi_{\text{AdvMPC-CRT}}$ protocol, realizing the corresponding $\mathcal{F}_{\text{AdvMPC-CRT}}$ functionality. This is presented in Section 3.4. Concretely, the first command implements a functionality that generates bounded randomness to accommodate computations that would otherwise wrap around in the original CRT representation. This construction is presented in Section 3.4.1 and the protocol that realizes uses the additional $\mathcal{F}_{\text{maBits}}$ command of the $\mathcal{F}_{\text{MPC-CRT}}$ functionality. The latter functionality facilitates the generation of multiply authenticated random bits [35]. Furthermore the consistency check that follows the Jacobi test requires certain computations to be performed over the integers. To realize this second command we need to convert a sharing from its CRT representation to an additive sharing of the CRT reconstructed value over the integers. We explain how to achieve this in Section 3.4.2.

3.2 Unauthenticated Arithmetic Black Box Functionality

$\mathcal{F}_{\text{ABBWithErrors}}$ (Fig. 3) is the functionality implementing an unauthenticated arithmetic black box MPC. Our Π_{RSAGen} protocol makes use of this functionality to perform a multiplication, in a more efficient manner than the actively-secure version. This does not cause the overall security of our protocol to depreciate, because the range in which the parties' inputs lie are implicitly checked when opening the product of the two sampled candidate primes, and the remaining primitives used in Π_{RSAGen} are actively secure.

For completeness, we detail the protocol realizing the unauthenticated ABB functionality, $\Pi_{\text{ABBWithErrors}}$, in Appendix C, Fig. 17. The $\Pi_{\text{ABBWithErrors}}$ protocol implements the online phase of the unauthenticated arithmetic black box, and it works in the $\mathcal{F}_{\text{ABBWithErrors-Prep}}$ -hybrid model. This functionality, realized by $\Pi_{\text{ABBWithErrors-Prep}}$, is used to generate the necessary preprocessing material for the online phase. Concretely, the required preprocessing material is (unauthenticated) input tuples and multiplication triples. The protocol for the preprocessing for tuples is listed in $\Pi_{\text{InputTuple}}$ (Fig. 18), while the protocol for the preprocessing of triples is listed in $\Pi_{\text{TripleGeneration}}$ (Fig. 19), in Appendix C. Note that the $\Pi_{\text{TripleGeneration}}$ protocol makes use of the standard $\mathcal{F}_{\text{Rand}}$ functionality, which is presented for completeness in Fig. 16, Appendix B.

For simplicity and clarity of presentation we describe here (and in Appendices C and B) the protocols implementing the standard unauthenticated arithmetic black box functionality. We also invoke the corresponding functionality in our protocol in the usual manner. However, we recommend this functionality to be implemented over a CRT representation of the sharings and inputs, meaning that we would need multiple MPC engines operating in parallel for each CRT component. We detail how to achieve the standard actively secure MPC functionality over CRT components in Section 3.3. The $\mathcal{F}_{\text{ABBWithErrors}}$ functionality can be also implemented over CRT components in the same manner. We have assumed the aforementioned implementation of the $\mathcal{F}_{\text{ABBWithErrors}}$ functionality for the efficiency analysis of our protocol.

3.3 MPC on CRT Components

In this subsection, we describe the functionality and the associated protocol to perform secure multiparty computation over a big composite modulus, by relying on the Chinese Remainder Theorem. The functionality $\mathcal{F}_{\text{MPC-CRT}}$ (Fig. 4) essentially implements the standard MPC functionality, but on sharings in their CRT representation. To accommodate computations on this type of sharings, we deploy ℓ MPC engines, for ℓ the maximum possible number of CRT components in the representation, as shown in $\Pi_{\text{MPC-CRT}}$ (Fig. 5). Each of these MPC engines operates over one of the prime moduli of the CRT representation, and each of these ℓ prime moduli is 128-bits long. All ℓ MPC engines operate in parallel, in a much smaller space than the big composite modulus over which the final reconstruction is performed. This has a profound impact on the efficiency of our Π_{RSAGen} protocol.

3.4 Advanced MPC CRT

The functionality $\mathcal{F}_{\text{MPC-CRT}}$ (Fig. 4) is similar to the *classic* MPC functionality, but over a direct product of finite fields. We also define $\mathcal{F}_{\text{AdvMPC-CRT}}$, which is the functionality $\mathcal{F}_{\text{MPC-CRT}}$ augmented with three additional commands. The first one is the *Rand2k* command, which samples a random secret shared value $r < 2^k$ in its CRT representation over ℓ' moduli, used in our Π_{RSAGen} protocol to ensure no overflows during computation. The second one is the *ConvInt* command, which allows the parties to convert a CRT sharing to an integer sharing of the same value. The third one is the *LevelUp* command, which extends the CRT representation of the sharings of the candidate primes p and q to the CRT representation of the same sharings, but with additional CRT components. This augmented functionality $\mathcal{F}_{\text{AdvMPC-CRT}}$ is described in Fig. 6. We subsequently detail how the three additional commands are realized.

3.4.1 Bounded Randomness in Shared CRT Representation. The *Rand2k* command allows us to sample a random CRT sharing, the reconstruction of which falls within a predetermined range. This is necessary in our main protocol to accommodate computations that would otherwise overflow over the initial (smaller) CRT representation. The protocol implementing the *Rand2k* command is listed in Π_{Rand2k} (Fig. 8), and it uses an additional command of the $\mathcal{F}_{\text{MPC-CRT}}$ functionality, namely the $\mathcal{F}_{\text{maBits}}$ command. The $\mathcal{F}_{\text{maBits}}$ command itself, presented in Fig. 7, is a slightly different version of the one presented by Rotaru et al. [35]. In our case, we modify the command so that it outputs the integer sharing of the bit, which was discarded in the original paper. We need this integer sharing later, in the Π_{ConvInt} protocol.

3.4.2 Converting a CRT Sharing to an Integer Sharing. Our main protocol requires a command which converts a CRT sharing to an integer sharing of the same underlying secret. This is necessary during the Jacobi test of Π_{RSAGen} , because we need to ensure that all computations in the exponent are performed over the integers, and hence the shares in the exponent are also reconstructed over the integers; otherwise, the correctness of the protocol is not guaranteed due to potential wrap around. To ensure that indeed the computations are done over the integers, we realize the *ConvInt* command with the protocol Π_{ConvInt} , listed in Fig. 9. This protocol allows the parties to obtain an unauthenticated integer

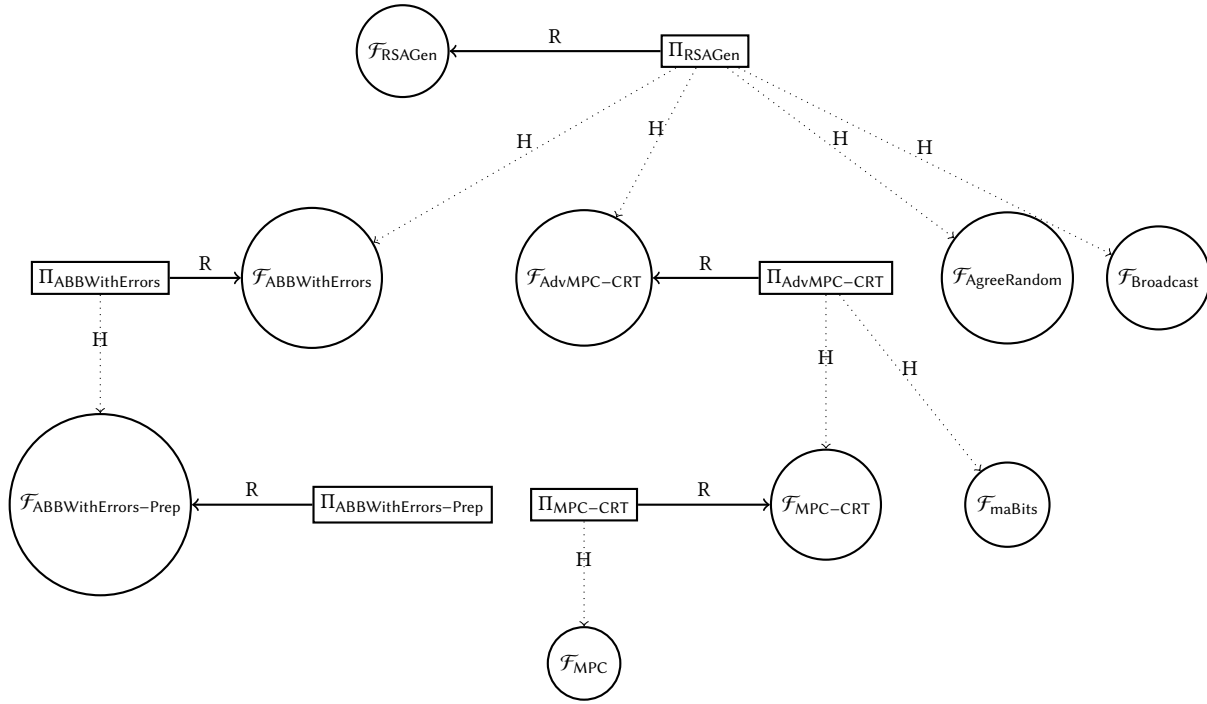


Figure 2: Functionality dependencies for RSA modulus generation.

Functionality $\mathcal{F}_{\text{ABBWithErrors}}$
Initialize: Parties call $\mathcal{F}_{\text{ABBWithErrors-Prep}}$ to receive preprocessing tuples and triples.
Input: Receive a value x from some party and store x .
Mult ($[x], [y]$): Await for Δ from the adversary. Compute $z = (x \cdot y) + \Delta$ and store $[z]$.
Share ($[x]$): For each corrupt party $i \in \mathcal{A}$ receive x_i from the adversary. Sample uniformly honest parties' shares $x_{j \notin \mathcal{A}}$ such that $\sum_{i=1}^n x_i = x$. Send x_i to P_i .

Figure 3: Arithmetic Black Box Functionality with Errors.

sharing of the CRT sharings they already hold, without leaking any information about the underlying secret value.

To see the correctness of the protocol Π_{ConvInt} observe the following:

- (1) For input values upper bounded by 2^B , the parties, invoking the protocol Π_{Rand2k} , sample shares of a random value upper bounded by $2^{B+\sigma}$. The additional σ bits of the random value ensure that, upon additive masking, the randomness statistically hides the input. The parties here receive two types of additive shares of the same random value: one CRT share, and the corresponding share over the integers.
- (2) In Step 2, the parties additively mask their input share, using the CRT share of the random value, open the result, and locally perform the CRT reconstruction of the masked value t .
- (3) In Step 3, the parties “unmask” their local value t , by subtracting the integer share of the randomness they possess from the local value t . Correctness follows because both the

Functionality $\mathcal{F}_{\text{MPC-CRT}}$
Let $[x, \ell]$ denote the identifiers for the ℓ components of the CRT representation a value x stored in the functionality. Let $A \subset \{1, \dots, n\}$ denote the index set of the corrupted parties.
Init: Receive p_1, \dots, p_ℓ primes from all parties, store them and compute $M_\ell = \prod_{i=1}^\ell p_i$.
Input: Receive a tuple $(x, \ell') \in \mathbb{Z}_{M_{\ell'}}$ with $\ell' \leq \ell$ from some party and store $([x, \ell'])$.
Add ($[x, \ell'], [y, \ell']$): Retrieve (x, ℓ') and (y, ℓ') from memory, compute $z = x + y \bmod M_{\ell'}$, and store $([z, \ell'])$.
Mult ($[x, \ell'], [y, \ell']$): Retrieve (x, ℓ') and (y, ℓ') from memory, compute $z = x \cdot y \bmod M_{\ell'}$, and store $([z, \ell'])$.
Open ($[x, \ell']$): Retrieve (x, ℓ') from memory and send the value x to all parties.
OpenTo ($[x, \ell'], j$): Retrieve (x, ℓ') from memory and send the CRT represented values (x, ℓ') to party P_j .

Figure 4: MPC over CRT Functionality.

CRT sharing of the randomness, and the integer sharing thereof reconstruct to the same value, and no wrap around can happen, due to the bound on the product of the CRT components, which is sufficiently large to accommodate the bounded input, the statistical security parameter, and the addition that is necessary to perform the masking.

The execution of the protocol Π_{ConvInt} does not leak any information about the secret. Indeed, the only opened value in the protocol

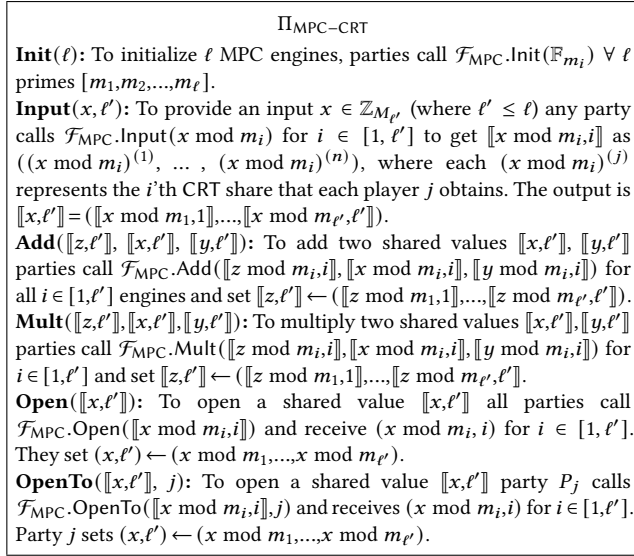


Figure 5: Protocol for arithmetic MPC over CRT moduli.

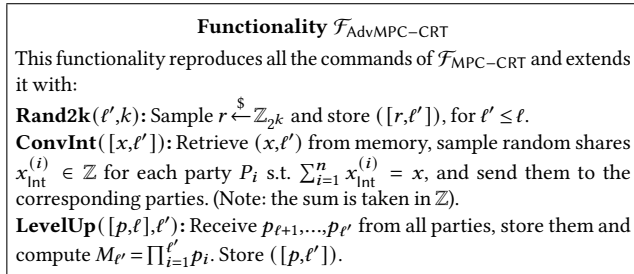


Figure 6: Advanced MPC over CRT Functionality.

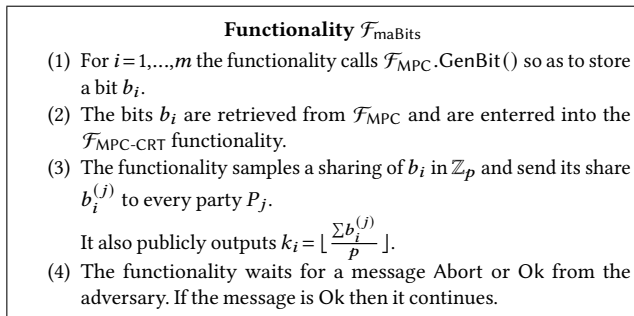


Figure 7: Random Bits Generation Functionality.

is $\llbracket x, \ell_{\text{Jac}} \rrbracket + \llbracket r, \ell_{\text{Jac}} \rrbracket$ with $r \xleftarrow{\$} [2^{B+\sigma}]$ and $x < 2^B$. Therefore, by Theorem 3.1, we have that the distance between the distribution of this opened value and the uniform distribution is upper bounded by $2^{-\sigma}$.

This protocol only produces an *unauthenticated* integer sharing, but the consistency of the shares is checked later in the Π_{RSAGen} protocol. During the broadcast at step 5 of the Consistency Check, a malicious adversary can broadcast any arbitrary value, but if the value of the shared secret would be altered by the adversary's broadcast, then the equality check which follows will fail with probability $1/2$.

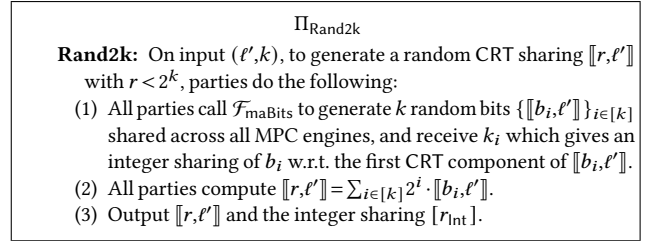
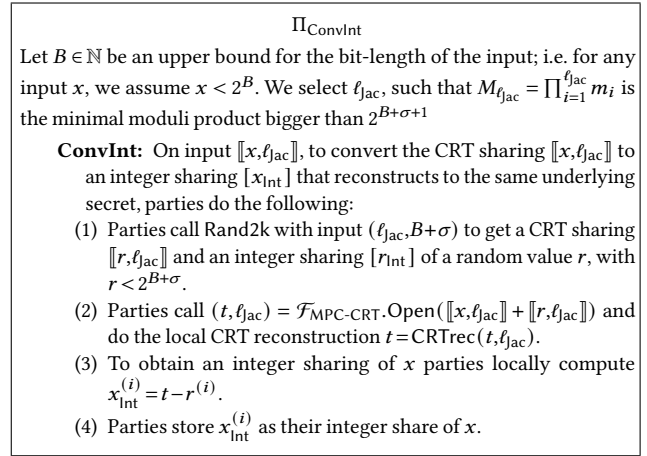
Figure 8: Protocol for generating a random CRT sharing $\llbracket r, \ell' \rrbracket$, which CRT reconstructs to a bounded random value $r < 2^k$, and the corresponding integer sharing $[r]_{\text{Int}}$.

Figure 9: Protocol for converting a bounded CRT sharing to an integer sharing.

THEOREM 3.1 ([36, APPENDIX A]). *Let M and K be positive integers, where $M \leq K$. Let the random variable X take values from $\{0, \dots, M-1\}$ and let the random variables U be uniform on $\{0, \dots, K-1\}$. Then $\Delta(U, X+U) \leq (M-1)/K$ is an upper bound for the distance between the two distributions.*

3.4.3 Extending the CRT Representation. The LevelUp command extends the CRT representation of $[p, \ell]$ and $[q, \ell]$, allowing us to compute over $M_{\ell'} > M_\ell$ for these two values. In our Π_{RSAGen} protocol, we use LevelUp command whenever a new operation on p and q could overflow the current CRT modulus. This happens twice: first during the consistency check, and then in the GCD test. We note that we execute this command only on $[p, \ell]$ and $[q, \ell]$, which we know to be bounded by $2^{\lambda+\sigma}$. We use this property in the Π_{LevelUp} protocol (Fig. 10), which implements the LevelUp command.

The protocol ensures that the new $\llbracket p, \ell' \rrbracket$ is a sharing of the same value as $\llbracket p, \ell \rrbracket$. Indeed, by sampling small enough values r and s at random, we can use r as a MAC key and s as a mask while avoiding any overflow. As in all MAC checks, a cheating adversary would have to guess r to successfully cheat. In addition, we make sure to only open $p \cdot r + s$ which is statistically close to uniform randomness because s is chosen uniformly random in $[0, 2^{\lambda+3 \cdot \sigma}]$ whereas $p \cdot r < 2^{\lambda+2 \cdot \sigma}$.

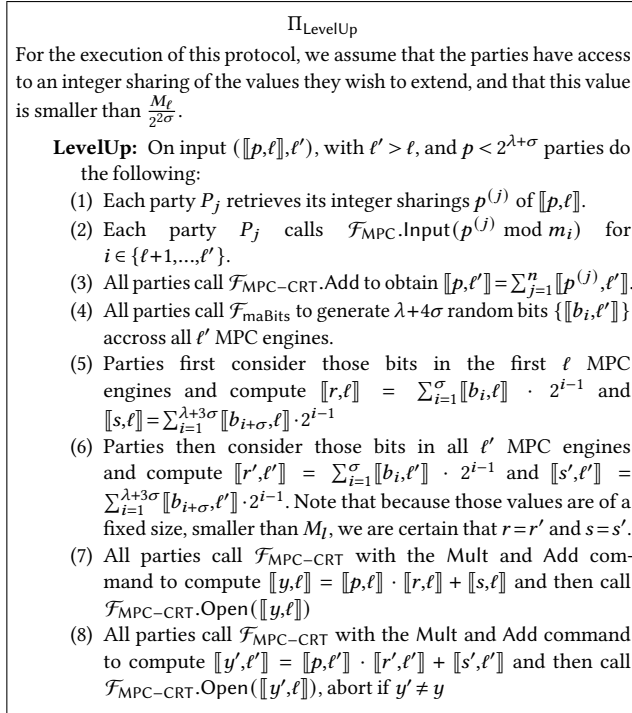


Figure 10: Protocol for extending the CRT representation of $\llbracket p, \ell \rrbracket$ and $\llbracket q, \ell \rrbracket$.

4 DISTRIBUTED GENERATION OF AN RSA BIPRIME

Our ideal functionality $\mathcal{F}_{\text{RSAGen}}$, listed in Fig. 11, consists of 5 steps: Sample, Combine, Jacobi, Consistency Check, and GCD Test. The first step, samples two candidate primes p and q of approximately 1024 bits each which are both coprime with M_{sample} , which significantly increases our chances of selecting a prime. The second step computes the product of the previously sampled candidate primes, and checks that it lies in the expected range (respecting the aforementioned bit-length), and that it is not coprime with M_{sample} . The third step follows the blueprint of Boneh and Franklin [5], checking biprimality in a way similar to Miller-Rabin primality testing, and returning Abort, if the product computed is not a biprime. The fourth step serves as a consistency check, confirming that all parties have input consistent shares in the so-called Jacobi step above. The last step is the GCD test, which catches some false positives that can potentially be introduced by the Jacobi test.

We now concretely detail the protocol Π_{RSAGen} (Fig. 12), realizing the $\mathcal{F}_{\text{RSAGen}}$ functionality. Each party P_j samples a multiplicative share $\hat{p}^{(j)}$, such that $\gcd(M_{\text{sample}}, \hat{p}^{(j)}) = 1$. The goal of the Sampling phase is to convert the multiplicative sharing $\hat{p} = \hat{p}^{(1)} \cdot \dots \cdot \hat{p}^{(n)}$ over the integers, into an additive sharing $p' = p'^{(1)} + \dots + p'^{(n)}$ over $\mathbb{Z}/(M_{\text{sample}}\mathbb{Z})$, such that $p' = \hat{p} \bmod M_{\text{sample}} = p'^{(1)} + \dots + p'^{(n)} = \hat{p}^{(1)} \cdot \dots \cdot \hat{p}^{(n)} \bmod M_{\text{sample}}$. So, each party P_j engages in a (semi-honest) multiplication with their secret share $\hat{p}^{(j)}$. In the end, all parties hold an additive sharing of the product $\hat{p}^{(j)} = \hat{p}_1^{(j)} + \dots + \hat{p}_n^{(j)} \bmod M_{\text{sample}}$. After the

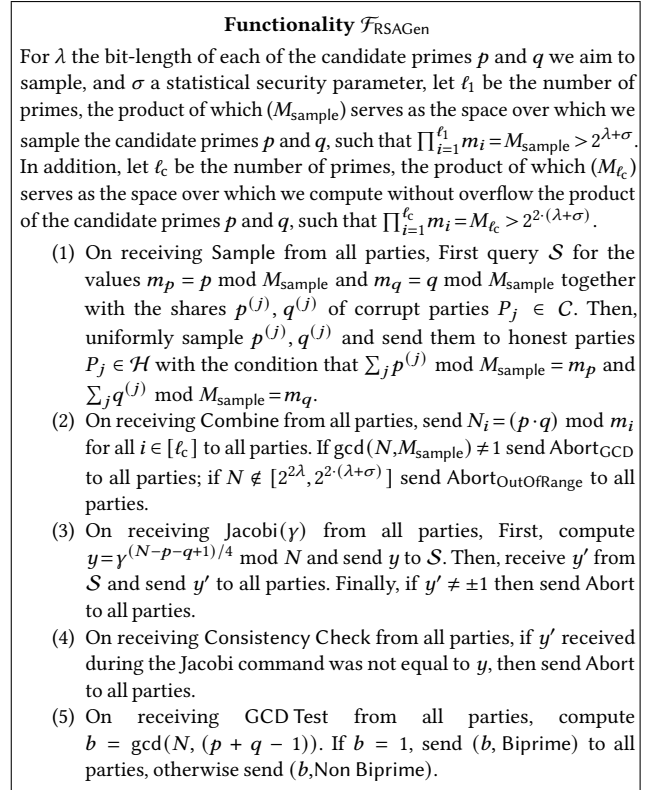


Figure 11: RSA Modulus Generation Functionality

multiplication is done over $\mathbb{Z}/(M_{\text{sample}}\mathbb{Z})$, parties set their local share as $p^{(j)} = p'^{(j)} + r^{(j)} \cdot M_{\text{sample}}$ and use this in the CRT Input procedure, over ℓ_c CRT components, and thus ℓ_c MPC engines.

Once two candidate primes p and q have been sampled in a secret shared fashion as described above, the Combine phase begins. First, the parties sum the contributions of each party into the additive sharing, over ℓ_c CRT components. Then, the candidate biprime is computed, using an actively secure multiplication over the CRT representation of the sharings of p and q . Lastly, the parties open the resulting candidate biprime N , and each party P_j locally performs the CRT reconstruction and obtains the biprime N in the standard form. Each party checks that the biprime respects the bounds in which it should lie, and that it is not coprime to the upper bound of the sampling range.

The parties then begin the biprimality testing with the Jacobi test, which needs to be repeated 128 times. The core of the Jacobi test we design offers passive security; to achieve active security, should the Jacobi test pass, we proceed with the Consistency Check phase. This step ensures that parties cannot go undetected, if they use inconsistent sharings in the Jacobi test. To realize this, first we need to increase our computing space to avoid potential overflows. We do that by means of the LevelUp command, which allows us to receive the same sharings in a CRT representation with additional CRT components (to accommodate the computations). Concretely, we extend from ℓ_c components of the combine step, to ℓ_{Jac} components, which suffice

for the correctness of the consistency check of the Jacobi. Then, using the Rand2k command, we receive bounded shared randomness in the CRT form with ℓ_{Jac} components. Using this randomness, we multiplicatively mask (guaranteed without overflow) the exponent of the Jacobi test, where the parties' shares have been contributed. This latter product is then converted from a CRT sharing with ℓ_{Jac} components to an integer sharing by calling the ConvInt command. The integer sharing is used to exponentiate the public value γ used in the Jacobi test, and it is then broadcasted. The randomness used in the masking operation is revealed, so that the parties can perform the final exponentiation of the Jacobi value computed to the power of the randomness in the clear. From the broadcasted values, the parties can also reconstruct again the masked version of the Jacobi test exponentiation. If the two latter values do not match, then some parties have input inconsistent shares, and the protocol aborts.

The last phase of our protocol is the GCD test, aiming at detecting (and discarding) any false positive biprimes that passed the Jacobi test. The GCD test is performed between the public biprime N , and the secret $\llbracket p+q-1 \rrbracket$, and if their GCD equals 1, the test passes. Let $Q_{\text{gcd}} > V \cdot N$, where $V = 2^{3\lambda+4\sigma}$. The goal is to output the product $a \cdot (p+q-1) + v \cdot N \bmod Q_{\text{gcd}}$, where $a \xleftarrow{\$} [N]$, and then perform the gcd computation between N and $a \cdot (p+q-1) + v \cdot N$ on public values. In our case v needs to statistically mask the product between a , which has $2(\lambda+\sigma)$ bits length, and $p+q-1$, which has $\lambda+\sigma$ bits length. Hence, $\log_2 v = 3(\lambda+\sigma) + \sigma$. Next, $M_{\ell_{\text{gcd}}}$ is computed, so that $v \cdot N$ fits Q_{gcd} , which makes $M_{\ell_{\text{gcd}}}$ to be $5\lambda+6\sigma$ bits long.

On a step by step basis, for the GCD test we use again the LevelUp command to extend the number of CRT components in our sharings of p and q . For the masking, similarly to the Jacobi test, we sample bounded randomness in CRT form with ℓ_{gcd} components, using again the Rand2k command. Before we open and reconstruct the final value \hat{z} , the gcd of which needs to be checked against the public biprime N , we also perform an additive masking with a bounded random value v . This ensures that no information about the sum of p and q , involved in the multiplicatively masking, can be factored out upon opening. Upon opening and reconstruction of the masked value, the final GCD test is performed, and if the open value is not coprime to N the protocol outputs abort and restarts.

THEOREM 4.1. *The execution of the protocol Π_{RSAGen} UC-securely realizes the functionality $\mathcal{F}_{\text{RSAGen}}$, in the $(\mathcal{F}_{\text{ABBWithErrors}}, \mathcal{F}_{\text{MPC}}, \mathcal{F}_{\text{AgreeRandom}}, \mathcal{F}_{\text{Broadcast}})$ -hybrid model with statistical security against a static, active adversary that corrupts up to $n-1$ parties.*

We refer the reader to Appendix A for the proof of Theorem 4.1.

5 PARAMETERS AND EFFICIENCY ANALYSIS

We generate biprimes of various bit-lengths, and hence security levels; namely $\lambda = \{1024, 1536, 2048\}$ as in the work of Chen et al. [7]. In the cases where a statistical security parameter σ needs to be considered, such as in the Sampling Phase, Jacobi test, masking and underlying MPC engines, we make sure to set $\sigma = 80$ to have a fair comparison with the analysis of Chen et al. [7], since they also used $\sigma = 80$, when measuring their concrete costs.

Given that our protocol requires several types of MPC engines, e.g., the ABBWithErrors, or the MPC-CRT, we use the MP-SPDZ framework [26] to get concrete communication costs for different

adversary structures. In the case of dishonest majority, we instantiate ABBWithErrors using the semi-honest version of the MASCOT protocol [27], whereas for the malicious case, which we need for building the MPC-CRT, we use LowGear [28], with TopGear [3] as the underlying ZK proof. For the 16 parties case, we use the HighGear protocol with the TopGear ZK-proof, which is also implemented in MP-SPDZ. The reason for choosing HighGear over LowGear is that for HighGear communication scales better in the number of parties.

We also give concrete costs for RSA-Sieve in the semi-honest, dishonest majority model. The only difference with the malicious case is that MPC-CRT can be instantiated with a cheaper protocol and no zero-knowledge proofs. For this variant, we use the classical SPDZ triple generation with no ZK proofs [6, 11], for which we get concrete costs by running the *hemi* protocol in MP-SPDZ [26]. The results are given in Table 2 for the two party case, while in Table 3 we have results for the 16 party case, where we also compare them with the the protocol of Chen et al. [7]. As it can be inferred from Table 2, for two parties, our protocol is a factor of 3.3-3.9 more communication-efficient than the state-of-the-art [7] in the semi-honest case, and by a factor of 32-37 in the malicious case (ranging for different bit-lengths of the birprimes generated). For the 16 party case, the protocol of Chen et al. [7] outperforms ours by a factor of approximately 2 in the semi-honest case. Then again, for the malicious case and for 16 parties, our protocol improves the communication cost over the state-of-the-art [7] by approximately 14-30 times.

In the following, we give an example of how we compute the cost using $\lambda = 1024$, in the dishonest majority case with malicious security. The number of primes used in the distributed sieving is fixed to 130, as in the work of Chen et al. [7], to achieve the same number of Sample iterations. Note that the product of the first 130 primes is 1019 bits long. Frankel et al. [16] select $r^{(j)}$ in the sampling phase at random from $[0, 2^n / M_{\text{sample}}]$ where n was the desired bit-length of p .

For λ being the bit-length of the candidate primes, we need to take their product in a space of double the size to avoid wrap around. Hence, M_{ℓ_c} , the product of primes in which the biprimes live, needs to be of length at least $2\lambda + 2\sigma$ bits, which results in $\ell_c = 18$ (i.e., we need 18 CRT components of 128 bits each). Similarly, we compute $\ell_{\text{Jac}} = 21$ and $\ell_{\text{gcd}} = 46$.

- (1) **Sampling phase.** The cost per semi-honest multiplication per party with ABBWithErrors is $(n-1)(128 \cdot k + k^2)$, where n is the number of parties and k is the field size [27]. Since the cost is quadratic in the field size, our ABBWithErrors will work over all the small primes composing M_{sample} . This brings the communication cost per triple at 17.027 kilobits with a total communication including the Beaver openings. The Input calls to ABBWithErrors in Step 2 amount to 0.264kbits. This makes steps 2 and 3 having a cost of 17.291 kbits. The remaining cost here comes from the Input calls to $\mathcal{F}_{\text{MPC-CRT}}$. This is instantiated using LowGear with TopGear as ZK proof, where the input tuple cost is 1.35 kbits for a 128-bit prime. This makes Step 7 in the Sampling phase amount to 48.67 kbits. One iteration of this phase has a total cost of 65.97 kbits.
- (2) **Combine.** The cost per multiplication triple using $\mathcal{F}_{\text{MPC-CRT}}$ amounts to 12.862 kbits per party. This brings the cost of one



Figure 12: RSA modulus generation protocol based on distributed sieving

execution of Step 2 to 231 kbits. The opening (Step 3) takes another 2.176 bits. One iteration of this phase has a total cost of 233 kbits.

- (3) **Jacobi test.** The cost of this phase is simply $\log_2(N) \cdot n$, which is $2n \cdot \lambda$ or about 4 kbits.
- (4) **Consistency check.** This phase begins with a call to LevelUp from ℓ_c to ℓ_{Jac} . The LevelUp protocol is ran twice, once for each candidate p and q . Concretely, per run, LevelUp requires $\ell_{\text{Jac}} - \ell_c = 3$ inputs per party, ℓ_{Jac} multiplications and ℓ_{Jac} openings on the CRT components. The more expensive part of LevelUp generates $\lambda + 4 \cdot \sigma = 1344$ maBits which amount to 22 Mbits, roughly the cost of one iteration of LevelUp. The entire cost of Step 1 in the consistency check is 44,551 kbits. Next, parties call Rand2k, which roughly costs σ random bit generations, concretely 2095 kbits. The multiplication cost is

simply $\ell_{\text{Jac}} \cdot 12.862 = 270.1$ kbits. The call to ConvInt requires a call to Rand2k and one opening, which amounts to 43,222 kbits. Finally, the parties need to broadcast an element in \mathbb{Z}_N and then open an element in all the ℓ_{Jac} CRT components, which requires communicating 4.8 kbits. One iteration of this phase has a total cost of 90,143 kbits.

- (5) **GCD test.** Here again we start with a call to LevelUp, which costs 56,558 kbits, and then a call to Rand2k for a cost of 45,477 kbits. Next is a multiplication on ℓ_{gcd} CRT components for a total cost of 591.7 kbits. Second to last, we do a final call to Rand2k with larger parameters, so the cost this time is 89,307 kbits. Finally, we open \hat{z} for 5.9 kbits. This phase thus requires a total of 191,940 kbits.

Scheme	[7]	Ours	[7]	Ours	[7]	Ours
κ	1024	1024	1536	1536	2048	2048
SH (MB)	139	41.68	416	116.55	910	243.30
ML (GB)	20.81	0.64	43.42	1.188	74.52	1.99

Table 2: Communication per party (two parties). For [7] the cost of the semi-honest protocol is based on the use of the OT extension of Keller et al. [27]. We consider this to be a fair comparison, as the sampling protocol is the major bottleneck and can be implemented using SilentOT. In our case the underlying MPC engine for sampling also used the same OT extension.

Scheme	[7]	Ours	[7]	Ours	[7]	Ours
κ	1024	1024	1536	1536	2048	2048
SH (GB)	2.09	4.34	6.24	12.17	13.65	25.23
ML (GB)	1020	68.8	4734	153.20	8100	281.91

Table 3: Communication per party malicious case (16 parties). For [7] the cost of the semi-honest protocol is based on the use of the OT extension of Keller et al. [27].

We present the detailed per-phase cost for 2 and 16 parties, and for $\lambda = \{1024, 1536, 2048\}$ in Table 4 for the malicious case, and in Table 5 for the semi-honest case.

Reducing the number of Input calls in generating bounded randomness. In the maBit protocol designed in [35], each random bit $\llbracket b \rrbracket$ produced in the main MPC engine producing randomness is later fed into the other MPC engines by every party calling Input command on a different sharing of b . By plugging their method directly into our Rand2k protocol, in order to generate n_B shared bits with n parties shared across ℓ engines will require $n_B \cdot n$ Input calls to each of the ℓ 128 bit prime MPC engines.

We can reduce the number of input calls by a factor of $\sim 128 - (\sigma + \log_2 m)$, where m is the batch size for generating maBits. The key insight is for parties to batch their bit shares instead of inputting them one by one. For example, if they want to batch 16 bit inputs at once, they can call $\mathcal{F}_{\text{MPC}}.\text{Input}(\sum_{k=0}^{15} 2^k b_k^{(j)} \bmod m_i)$, where $b_k^{(j)}$ would be party P_j 's share of the k^{th} bit in the maBit protocol. We need to take into account now that the random linear combination at the end is done over slightly larger secrets (16 bits instead of a single one), so we need to increase the random coefficients by 16 bits in order for the security reduction to go through easily. The proof of this small optimization is relatively straightforward, since one can use this as an oracle to solve the MSSP problem described in [35] by simply scaling the random coefficients. To fit everything in a 128-bit prime MPC engine, we pack 16 inputs together, while maintaining a maBit batch of 2^{15} bits produced at once.

The honest majority case with active security. Since our protocol works with any actively secure protocol, where the secret reconstruction is linear, we can instantiate it with the most efficient protocols for MPC for large field arithmetic [9]. The cost analysis of such an instantiation can be seen in Table 6.

κ	1024		1536		2048	
n	2	16	2	16	2	16
per-phase cost for one instance (Megabits)						
Sieving	0.36	51.2	0.5	73.5	0.68	95.7
BP test	0.004	0.03	0.006	0.04	0.008	0.06
Check	46.14	4344	68.42	8166	93.4	13185
expected cost to sample a biprime (GBytes)						
E[Iter.]	3607	3607	7251	7251	11832	11832
E[Total]	0.64	68.8	1.18	153.2	1.99	281.91

Table 4: Communication per party: malicious case. The GCD test is included in E[Total], as that is an one-time cost. Check step happens σ times.

κ	1024		1536		2048	
n	2	16	2	16	2	16
per-phase cost for one instance (kilobits)						
Sieving	82.97	9391	118.10	13175	152.44	16784
BP test	4.096	32	6.144	49.152	8.192	65.536
expected cost to sample a biprime (megabytes)						
E[Iter.]	3607	3607	7251	7251	11832	11832
E[Total]	41.68	4346	116.55	12173	243.3	25230

Table 5: Communication per party: semi-honest case.

κ	1024	1536	2048
megabytes	107.62	227.41	408.57

Table 6: Communication per party: malicious honest majority case (3 parties).

ACKNOWLEDGEMENTS

We thank Ivan Damgård and Muthu Venkitasubramaniam for their helpful discussions and the anonymous reviewers for their comments and observations. This work has been supported in part by ERC Advanced Grant ERC-2015-AdG-IMPACT, by the Defense Advanced Research Projects Agency (DARPA) under contract No. HR001120C0085, and by CyberSecurity Research Flanders with reference number VR20192203. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the ERC, DARPA, the U.S. Government or Cyber Security Research Flanders. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

REFERENCES

- [1] Joy Algesheimer, Jan Camenisch, and Victor Shoup. 2002. Efficient Computation Modulo a Shared Secret with Application to the Generation of Shared Safe-Prime Products. In *CRYPTO 2002 (LNCS, Vol. 2442)*, Moti Yung (Ed.). Springer, Heidelberg,

- 417–432. https://doi.org/10.1007/3-540-45708-9_27
- [2] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *ACM CCS 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, 805–817. <https://doi.org/10.1145/2976749.2978331>
- [3] Carsten Baum, Daniele Cozzo, and Nigel P. Smart. 2019. Using TopGear in Overdrive: A More Efficient ZKPoK for SPDZ. In *SAC 2019 (LNCS, Vol. 11959)*, Kenneth G. Paterson and Douglas Stebila (Eds.). Springer, Heidelberg, 274–302. https://doi.org/10.1007/978-3-030-38471-5_12
- [4] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. 2018. Verifiable Delay Functions. In *CRYPTO 2018, Part I (LNCS, Vol. 10991)*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer, Heidelberg, 757–788. https://doi.org/10.1007/978-3-319-96884-1_25
- [5] Dan Boneh and Matthew K. Franklin. 1997. Efficient Generation of Shared RSA Keys (Extended Abstract). In *CRYPTO'97 (LNCS, Vol. 1294)*, Burton S. Kaliski Jr. (Ed.). Springer, Heidelberg, 425–439. <https://doi.org/10.1007/BFb0052253>
- [6] Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. 2020. MOTION - A Framework for Mixed-Protocol Multi-Party Computation. Cryptology ePrint Archive, Report 2020/1137. <https://eprint.iacr.org/2020/1137>
- [7] Megan Chen, Ran Cohen, Jack Doerner, Yashvanth Kondi, Eysa Lee, Schuyler Rosefield, and abhi shelat. 2020. Multiparty Generation of an RSA Modulus. In *CRYPTO 2020, Part III (LNCS, Vol. 12172)*, Daniele Micciancio and Thomas Ristenpart (Eds.). Springer, Heidelberg, 64–93. https://doi.org/10.1007/978-3-030-56877-1_3
- [8] Megan Chen, Carmit Hazay, Yuval Ishai, Yuriy Kashnikov, Daniele Micciancio, Tarik Riviere, Abhi Shelat, Muthuramakrishnan Venkatasubramanian, and Ruihan Wang. 2020. Diogenes: Lightweight Scalable RSA Modulus Generation with a Dishonest Majority. *IACR Cryptol. ePrint Arch.* 2020 (2020), 374.
- [9] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. 2018. Fast Large-Scale Honest-Majority MPC for Malicious Adversaries. In *CRYPTO 2018, Part III (LNCS, Vol. 10993)*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer, Heidelberg, 34–64. https://doi.org/10.1007/978-3-319-96878-0_2
- [10] Anders P. K. Dalskov, Claudio Orlandi, Marcel Keller, Kris Shrivash, and Haya Shulman. 2020. Securing DNSSEC Keys via Threshold ECDSA from Generic MPC. In *ESORICS 2020, Part II (LNCS, Vol. 12309)*, Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider (Eds.). Springer, Heidelberg, 654–673. https://doi.org/10.1007/978-3-030-59013-0_32
- [11] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. 2013. Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits. In *ESORICS 2013 (LNCS, Vol. 8134)*, Jason Crampton, Sushil Jajodia, and Keith Myers (Eds.). Springer, Heidelberg, 1–18. https://doi.org/10.1007/978-3-642-40203-6_1
- [12] Ivan Damgård and Gert Læssøe Mikkelsen. 2010. Efficient, Robust and Constant-Round Distributed RSA Key Generation. In *TCC 2010 (LNCS, Vol. 5978)*, Daniele Micciancio (Ed.). Springer, Heidelberg, 183–200. https://doi.org/10.1007/978-3-642-11799-2_12
- [13] Yvo Desmedt. 1994. Threshold cryptography. *European Transactions on Telecommunications* 5, 4 (July/Aug. 1994), 449–457.
- [14] Yvo Desmedt. 1998. Some Recent Research Aspects of Threshold Cryptography (Invited Lecture). In *ISW'97 (LNCS, Vol. 1396)*, Eiji Okamoto, George I. Davida, and Masahiro Mambo (Eds.). Springer, Heidelberg, 158–173.
- [15] Yvo Desmedt and Yair Frankel. 1990. Threshold Cryptosystems. In *CRYPTO'89 (LNCS, Vol. 435)*, Gilles Brassard (Ed.). Springer, Heidelberg, 307–315. https://doi.org/10.1007/0-387-34805-0_28
- [16] Yair Frankel, Philip D. MacKenzie, and Moti Yung. 1998. Robust Efficient Distributed RSA-Key Generation. In *17th ACM PODC*, Brian A. Coan and Yehuda Afek (Eds.). ACM, 320. <https://doi.org/10.1145/277697.277779>
- [17] Tore Kasper Frederiksen, Yehuda Lindell, Valery Osheter, and Benny Pinkas. 2018. Fast Distributed RSA Key Generation for Semi-honest and Malicious Adversaries. In *CRYPTO 2018, Part II (LNCS, Vol. 10992)*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer, Heidelberg, 331–361. https://doi.org/10.1007/978-3-319-96881-0_12
- [18] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. 2017. High-Throughput Secure Three-Party Computation for Malicious Adversaries and an Honest Majority. In *EUROCRYPT 2017, Part II (LNCS, Vol. 10211)*, Jean-Sébastien Coron and Jesper Buus Nielsen (Eds.). Springer, Heidelberg, 225–255. https://doi.org/10.1007/978-3-319-56614-6_8
- [19] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. 1998. Simplified VSS and Fast-Track Multiparty Computations with Applications to Threshold Cryptography. In *17th ACM PODC*, Brian A. Coan and Yehuda Afek (Eds.). ACM, 101–111. <https://doi.org/10.1145/277697.277716>
- [20] Niv Gilboa. 1999. Two Party RSA Key Generation. In *CRYPTO'99 (LNCS, Vol. 1666)*, Michael J. Wiener (Ed.). Springer, Heidelberg, 116–129. https://doi.org/10.1007/3-540-48405-1_8
- [21] Lorenzo Grassi, Christian Rechberger, Dragos Rotaru, Peter Scholl, and Nigel P. Smart. 2016. MPC-Friendly Symmetric Key Primitives. In *ACM CCS 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, 430–443. <https://doi.org/10.1145/2976749.2978332>
- [22] Carmit Hazay, Gert Læssøe Mikkelsen, Tal Rabin, and Tomas Toft. 2012. Efficient RSA Key Generation and Threshold Paillier in the Two-Party Setting. In *CT-RSA 2012 (LNCS, Vol. 7178)*, Orr Dunkelman (Ed.). Springer, Heidelberg, 313–331. https://doi.org/10.1007/978-3-642-27954-6_20
- [23] Carmit Hazay, Gert Læssøe Mikkelsen, Tal Rabin, Tomas Toft, and Angelo Agatino Nicolosi. 2019. Efficient RSA Key Generation and Threshold Paillier in the Two-Party Setting. *Journal of Cryptology* 32, 2 (April 2019), 265–323. <https://doi.org/10.1007/s00145-017-9275-7>
- [24] Lukas Helming, Daniel Kales, Sebastian Ramacher, and Roman Walch. 2020. Multi-Party Revocation in Sovrin: Performance through Distributed Trust. Cryptology ePrint Archive, Report 2020/724. <https://eprint.iacr.org/2020/724>
- [25] Jonathan Katz and Yehuda Lindell. 2020. *Introduction to Modern Cryptography*. CRC press.
- [26] Marcel Keller. 2020. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1575–1590.
- [27] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2016. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. In *ACM CCS 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, 830–842. <https://doi.org/10.1145/2976749.2978357>
- [28] Marcel Keller, Valerio Pastro, and Dragos Rotaru. 2018. Overdrive: Making SPDZ Great Again. In *EUROCRYPT 2018, Part III (LNCS, Vol. 10822)*, Jesper Buus Nielsen and Vincent Rijmen (Eds.). Springer, Heidelberg, 158–189. https://doi.org/10.1007/978-3-319-78372-7_6
- [29] Michael Malkin, Thomas D. Wu, and Dan Boneh. 1999. Experimenting with Shared Generation of RSA Keys. In *NDSS'99*. The Internet Society.
- [30] Krzysztof Pietrzak. 2019. Simple Verifiable Delay Functions. In *ITCS 2019*, Avrim Blum (Ed.), Vol. 124. LIPIcs, 60:1–60:15. <https://doi.org/10.4230/LIPIcs.ITCS.2019.60>
- [31] Guillaume Poupard and Jacques Stern. 1998. Generation of Shared RSA Keys by Two Parties. In *ASIACRYPT'98 (LNCS, Vol. 1514)*, Kazuo Ohta and Dingyi Pei (Eds.). Springer, Heidelberg, 11–24. https://doi.org/10.1007/3-540-49649-1_2
- [32] Tal Rabin. 1998. A Simplified Approach to Threshold and Proactive RSA. In *CRYPTO'98 (LNCS, Vol. 1462)*, Hugo Krawczyk (Ed.). Springer, Heidelberg, 89–104. <https://doi.org/10.1007/BFb0055722>
- [33] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. 1978. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the Association for Computing Machinery* 21, 2 (1978), 120–126.
- [34] J. Barkley Rosser and Lowell Schoenfeld. 1962. Approximate formulas for some functions of prime numbers. *Illinois Journal of Mathematics* 6, 1 (1962), 64–94. <https://doi.org/10.1215/ijm/1255631807>
- [35] Dragos Rotaru, Nigel P. Smart, Titouan Tanguy, Frederik Vercauteren, and Tim Wood. 2019. Actively Secure Setup for SPDZ. Cryptology ePrint Archive, Report 2019/1300. <https://eprint.iacr.org/2019/1300>
- [36] Berry Schoenmakers and Pim Tuyls. 2006. Efficient Binary Conversion for Paillier Encrypted Values. In *EUROCRYPT 2006 (LNCS, Vol. 4004)*, Serge Vaudenay (Ed.). Springer, Heidelberg, 522–537. https://doi.org/10.1007/11761679_31
- [37] Adi Shamir. 1979. How to Share a Secret. *Communications of the Association for Computing Machinery* 22, 11 (Nov. 1979), 612–613.
- [38] Nigel P. Smart and Younes Talibi Alaoui. 2019. Distributing Any Elliptic Curve Based Protocol. In *17th IMA International Conference on Cryptography and Coding (LNCS, Vol. 11929)*, Martin Albrecht (Ed.). Springer, Heidelberg, 342–366. https://doi.org/10.1007/978-3-030-35199-1_17
- [39] Benjamin Wesolowski. 2019. Efficient Verifiable Delay Functions. In *EUROCRYPT 2019, Part III (LNCS, Vol. 11478)*, Yuval Ishai and Vincent Rijmen (Eds.). Springer, Heidelberg, 379–407. https://doi.org/10.1007/978-3-030-17659-4_13

A PROOF OF THEOREM 4.1

Proof Sketch. Let \mathcal{A} be a static malicious adversary, who interacts with the parties running Π_{RSAGen} and can corrupt up to $n-1$ parties. We construct a simulator \mathcal{S} , simulating the ideal functionality $\mathcal{F}_{\text{RSAGen}}$, such that no environment \mathcal{Z} can distinguish whether it is interacting with \mathcal{A} and the Π_{RSAGen} , or with \mathcal{A} and $\mathcal{F}_{\text{RSAGen}}$. Let \mathcal{C} denote the set of (up to $n-1$) corrupted parties and let \mathcal{H} denote the set of honest parties. The simulator \mathcal{S} proceeds as follows:

Sample: The simulator performs all the steps below in parallel for p and q .

- (1) For each honest $P_j \in \mathcal{H}$, \mathcal{S} samples $\hat{p}^{(j)}$ such that $\gcd(M_{\text{sample}}, \hat{p}^{(j)}) = 1$.

- (2) For each honest $P_j \in \mathcal{H}$, \mathcal{S} calls $\mathcal{F}_{\text{ABBWithErrors}}.\text{Input}(\hat{p}^{(j)})$. For each corrupt $P_j \in \mathcal{C}$, \mathcal{S} receives $\mathcal{F}_{\text{ABBWithErrors}}.\text{Input}(\hat{p}^{(j)})$ from \mathcal{A} .
- (3) When all parties call $\mathcal{F}_{\text{ABBWithErrors}}.\text{Mult}$, \mathcal{S} waits for Δ_p from \mathcal{A} . After receiving Δ_p , \mathcal{S} computes $p' = \Delta_p + \prod_{j=1}^n \hat{p}^{(j)}$.
- (4) When all parties call $\mathcal{F}_{\text{ABBWithErrors}}.\text{Share}(p')$, \mathcal{S} receives from \mathcal{A} the shares $p'^{(j)}$ for each corrupt $P_j \in \mathcal{C}$. It then samples and stores the remaining shares $p'^{(j)}$ for honest $P_j \in \mathcal{H}$ such that $p' = \sum_{j=1}^n p'^{(j)}$.
- (5) For each honest $P_j \in \mathcal{H}$, \mathcal{S} samples an appropriate $r^{(j)}$ such that $p^{(j)} = p'^{(j)} + r^{(j)} \cdot M_{\text{sample}}$ lies in the range $[2^\lambda, 2^{\lambda+\sigma}]$.
- (6) When each party calls $\mathcal{F}_{\text{MPC-CRT}}.\text{Input}(p^{(j)}, \ell_c)$ in Step 7, \mathcal{S} receives from \mathcal{A} the inputs $(p^{(j)}, \ell_c)$ for each corrupt $P_j \in \mathcal{C}$. With these, \mathcal{S} can reconstruct $p^{(j)}$ for each corrupt P_j and then compute $m_p = \sum_j p^{(j)} \bmod M_{\text{sample}}$ using also its simulated shares. It then sends Sample to $\mathcal{F}_{\text{RSAGen}}$ on behalf of the corrupt parties and, when prompted, submits m_p and the $p^{(j)}$ that it reconstructed. To continue simulating the protocol, \mathcal{S} inputs its own simulated $p^{(j)}$ into $\mathcal{F}_{\text{MPC-CRT}}$ on behalf of the honest parties $P_j \in \mathcal{H}$.

By computing the residue of p and q modulo M_{sample} as influenced by \mathcal{A} in Step 6 of the protocol, \mathcal{S} ensures that the distribution of $N \bmod M_{\text{sample}}$ produced by $\mathcal{F}_{\text{RSAGen}}$ is identical to the one in the protocol. At this stage of the protocol, there is no transcript for \mathcal{S} to simulate as the parties have only executed calls to other functionalities. We also note that the simulated shares $p^{(j)}$ are statistically close to the random shares sampled by $\mathcal{F}_{\text{RSAGen}}$, as measured by Lemma A.1, and identically distributed to the honest shares in a real execution.

Combine:

- (1) When all parties call $\mathcal{F}_{\text{MPC-CRT}}.\text{Sum}(\llbracket p, \ell_c \rrbracket, \llbracket p^{(1)}, \ell_c \rrbracket, \dots, \llbracket p^{(n)}, \ell_c \rrbracket)$ and $\mathcal{F}_{\text{MPC-CRT}}.\text{Sum}(\llbracket q, \ell_c \rrbracket, \llbracket q^{(1)}, \ell_c \rrbracket, \dots, \llbracket q^{(n)}, \ell_c \rrbracket)$ in Step 1, \mathcal{S} internally executes the corresponding MPC sums.
- (2) When all parties call $\mathcal{F}_{\text{MPC-CRT}}.\text{Mult}(\llbracket N, \ell_c \rrbracket, \llbracket p, \ell_c \rrbracket, \llbracket q, \ell_c \rrbracket)$ in Step 2, \mathcal{S} internally executes the corresponding MPC multiplications.
- (3) When all parties call $\mathcal{F}_{\text{MPC-CRT}}.\text{Open}(\llbracket N, \ell_c \rrbracket)$ in Step 3, \mathcal{S} sends Combine to $\mathcal{F}_{\text{RSAGen}}$ on behalf of the corrupt parties. Once the honest parties also send Combine to the functionality, \mathcal{S} receives N_i for all $i \in [\ell_c]$. To simulate the $\mathcal{F}_{\text{MPC-CRT}}.\text{Open}$ instruction, \mathcal{S} then sends the N_i values it received to each corrupt party. \mathcal{S} also updates its internal simulations of the $\mathcal{F}_{\text{MPC-CRT}}$ instances so that they hold the correct values for N .
- (4) If \mathcal{S} receives $\text{Abort}_{\text{GCD}}$ or $\text{Abort}_{\text{OutOfRange}}$ from $\mathcal{F}_{\text{RSAGen}}$, it makes the simulated honest parties also output the corresponding Abort in the protocol.

As the shares input by \mathcal{A} are passed on $\mathcal{F}_{\text{RSAGen}}$ for the generation of N , and as the shares simulated by \mathcal{S} are statistically close to those sampled at random by $\mathcal{F}_{\text{RSAGen}}$, the distribution of N output by $\mathcal{F}_{\text{RSAGen}}$ is statistically close to the one produced by \mathcal{S} , which is itself identically distributed to those of a real execution.

Jacobi:

- (1) When parties call $\mathcal{F}_{\text{AgreeRandom}}$, \mathcal{S} simulates the sampling of the public y .

- (2) The simulator then queries $\mathcal{F}_{\text{RSAGen}}.\text{Jacobi}(y)$ and receives y .
- (3) To simulate the broadcast calls, \mathcal{S} samples r_j at random to compute $y^{(j)} = y^{r_j}$ for the honest parties $P_j \in \mathcal{H}$ and then modifies one of these shares $y^{(i)}$ for $P_i \in \mathcal{H}$ such that $y^{(i)} = y \cdot (\prod_{j \neq i} y^{(j)})^{-1}$. Here, to compute $y^{(j)}$ for the corrupt parties $P_j \in \mathcal{C}$, the simulator uses the shares $p^{(j)}, q^{(j)}$ that \mathcal{A} input to $\mathcal{F}_{\text{ABBWithErrors}}$ during sampling. Then \mathcal{S} uses these simulated honest $y^{(j)}$'s as the broadcast values of the honest parties.
- (4) When the corrupt parties call $\mathcal{F}_{\text{Broadcast}}(y^{(j)})$, \mathcal{S} computes the new value of $y' = \prod_j y^{(j)}$ and sends it to $\mathcal{F}_{\text{RSAGen}}$.
- (5) If \mathcal{S} receives Abort from $\mathcal{F}_{\text{RSAGen}}$, it makes the simulated honest parties output Abort.

Since the simulated $p^{(j)}$ and $q^{(j)}$ values that \mathcal{S} holds for $P_j \in \mathcal{H}$ are statistically close to uniform, the distribution of the broadcast $y^{(j)}$ values are statistically close to the protocol and consistent with the correct Jacobi test result first output by $\mathcal{F}_{\text{RSAGen}}$. If \mathcal{A} cheats by using inconsistent values during its broadcast, then \mathcal{S} correctly updates the result of the Jacobi test by passing the new y' to $\mathcal{F}_{\text{RSAGen}}$.

Consistency Check:

- (1) When all parties call $\mathcal{F}_{\text{AdvMPC-CRT}}.\text{Rand2k}$, \mathcal{S} samples a random $x < 2^\sigma$ and receives the shares $(x^{(j)}, \ell_{\text{Jac}})$ for each corrupt $P_j \in \mathcal{C}$ from \mathcal{A} . It then samples the remaining shares $(x^{(j)}, \ell_{\text{Jac}})$ for honest $P_j \in \mathcal{H}$, such that $x = \text{CRTrec}(x, \ell_1) = \sum_{j=1}^n (x^{(j)}, \ell_{\text{Jac}})$.
- (2) When all parties call $\mathcal{F}_{\text{MPC-CRT}}.\text{Mult}(\llbracket t, \ell_{\text{Jac}} \rrbracket, \llbracket x, \ell_{\text{Jac}} \rrbracket, \llbracket ((N-p-q+1)/4), \ell_{\text{Jac}} \rrbracket)$, \mathcal{S} internally executes the MPC multiplication.
- (3) When all parties call $\mathcal{F}_{\text{AdvMPC-CRT}}.\text{ConvInt}(\llbracket t, \ell_{\text{Jac}} \rrbracket)$, \mathcal{S} samples $t_{\text{Int}}^{(j)}$ for each party P_j such that $t = \sum_{j=1}^n t_{\text{Int}}^{(j)}$, and sends them.
- (4) To simulate the broadcast calls, \mathcal{S} modifies one of the honest shares $y^{t_{\text{Int}}^{(i)}}$ for $P_i \in \mathcal{H}$ such that $y^{t_{\text{Int}}^{(i)}} = y^x \cdot (\prod_{j \neq i} y^{t_{\text{Int}}^{(j)}})^{-1}$, where y is the value given to \mathcal{S} by $\mathcal{F}_{\text{RSAGen}}$ during the Jacobi command and where \mathcal{S} uses its internal values of x and $p^{(j)}, q^{(j)}$ to compute $y^{t_{\text{Int}}^{(j)}}$ of the corrupt parties. \mathcal{S} then broadcasts $y^{t_{\text{Int}}^{(j)}}$ on behalf of the honest parties.
- (5) When all parties call $\mathcal{F}_{\text{MPC-CRT}}.\text{Open}(\llbracket x, \ell_{\text{Jac}} \rrbracket)$, \mathcal{S} simulates the opening and sends Consistency Check to $\mathcal{F}_{\text{RSAGen}}$ on behalf of the corrupt parties.
- (6) To reply to $\mathcal{F}_{\text{RSAGen}}$ about the abort, \mathcal{S} checks whether $\prod_j y^{t_{\text{Int}}^{(j)}} = y^x$ using the values that were broadcast. If the equality fails, \mathcal{S} sends Abort to \mathcal{A} on behalf of the honest parties.

The values used by \mathcal{S} in Step 2 are identically distributed to those in the protocol, so the distribution of the modified share $y^{t_{\text{Int}}^{(i)}}$ in Step 4 is statistically close, as measured by Lemma A.1. As x is sampled at random identically, and the influence of \mathcal{A} in the integer conversion of t is preserved, then the distribution of the broadcast of Step 4 is statistically close to the distribution of a real transcript.

Finally, the probability of abort when \mathcal{A} behaves honestly remains the same, since Step 4 modifies the honest shares to be consistent with the y output. When \mathcal{A} acts maliciously in the broadcast of $y^{t_{\text{Int}}^{(j)}}$, we claim that it has a negligible chance of making the equality hold, if it had already cheated in the Jacobi test. If it successfully makes the

equality hold, this creates a difference between real and ideal world as $\mathcal{F}_{\text{RSA Gen}}$ would abort, since it received a modified y' from \mathcal{S} but \mathcal{S} would not abort. As we assume that \mathcal{A} successfully cheated in the Jacobi test, we can assume that $\tilde{N} = (N - p - q + 1)/4 \neq 0 \pmod{\phi(N)}$ and that the adversary introduced an error Δ_j such that $y' = y^{\tilde{N} + \Delta_j} = \pm 1 \pmod{N}$. When the adversary cheats in the broadcast of the consistency check, the simulator computes $y^{t_{\text{int}} + \Delta_t}$, where $\Delta_t \neq 0$ represents the error introduced by \mathcal{A} . Thus, for the equality to hold, \mathcal{A} needs to commit to Δ_t during the broadcast, such that $y^{x\tilde{N} + \Delta_t} = y^x = y^{x\tilde{N}} \pmod{N}$. Since the distribution of $x\tilde{N}$ is uniform with a min-entropy of $2^{-\sigma}$, because of the sampling of x , \mathcal{A} has probability at most $2^{-\sigma+1}$ of finding the correct Δ_t . We finally note that Δ_t cannot simply be the right value in the group of exponents with higher probability than this, because Rosser and Schoenfeld [34] showed that

$$\phi(N) > \frac{N}{e^{\gamma} \log \log N},$$

where γ is the Euler–Mascheroni constant. This gives us that $\log_2(\phi(N)) > \sigma$, when $\log_2(N) > 2048$ and $\sigma \sim 80$.

GCD Test:

- (1) When all parties call $\mathcal{F}_{\text{AdvMPC-CRT.Rand2k}}$ in Step 2, \mathcal{S} samples a random $a < 2^{2\lambda+2\sigma}$ and receives the shares $(a^{(j)}, \ell_{\text{gcd}})$ for each corrupt $P_j \in C$ from \mathcal{A} . It then samples the remaining shares $(a^{(j)}, \ell_{\text{gcd}})$ for honest $P_j \in \mathcal{H}$, such that $a = \text{CRTrec}(a, \ell_c) = \sum_{j=1}^n (a^{(j)}, \ell_{\text{gcd}})$, and stores them.
- (2) When all parties call $\mathcal{F}_{\text{MPC-CRT.Mult}}(\llbracket z, \ell_{\text{gcd}} \rrbracket, \llbracket a, \ell_{\text{gcd}} \rrbracket, \llbracket (p+q-1), \ell_{\text{gcd}} \rrbracket)$, \mathcal{S} internally executes the MPC multiplication.
- (3) When all parties call $\mathcal{F}_{\text{AdvMPC-CRT.Rand2k}}$ in Step 4, \mathcal{S} samples a random $v < 2^{3\lambda+4\sigma}$ and receives the shares $(v^{(j)}, \ell_{\text{gcd}})$ for each corrupt $P_j \in C$ from \mathcal{A} . It then samples the remaining shares $(v^{(j)}, \ell_{\text{gcd}})$ for honest $P_j \in \mathcal{H}$, such that $v = \text{CRTrec}(v, \ell_c) = \sum_{j=1}^n (v^{(j)}, \ell_{\text{gcd}})$, and stores them.
- (4) Before opening, \mathcal{S} queries $\mathcal{F}_{\text{RSA Gen.GCD Test}}$ and receives $b = \text{gcd}(N, p+q-1)$. It then computes $b' = \text{gcd}(N, a)$ and samples a new \tilde{z} subject to the condition that $\text{gcd}(N, \tilde{z}) = \max\{b, b'\}$. It finally replaces \hat{z} by the new value $\hat{z} = \tilde{z} + v \cdot N$.
- (5) When all parties call $\mathcal{F}_{\text{MPC-CRT.Open}}(\llbracket \hat{z}, \ell_{\text{gcd}} \rrbracket)$, \mathcal{S} simulates the opening using the modified \hat{z} and outputs Abort if $\mathcal{F}_{\text{RSA Gen}}$ output Non Biprime.

If $\text{gcd}(N, p+q-1) = b$ then $b \mid z + v \cdot N$ and $b \mid z = a(p+q-1)$. Now, $\text{gcd}(N, \hat{z})$ in the protocol can differ from $b = \text{gcd}(N, p+q-1)$, if $\text{gcd}(N, a) = b' > b$; thus, by sampling a random \tilde{z} such that $\text{gcd}(N, \tilde{z}) = \max\{b, b'\}$, the simulator remains consistent with both b and the probability that $\text{gcd}(N, a) = b' > b$ occurs, since a is sampled identically. By adding a sufficiently random v to z any information about $p+q-1$ other than b is masked, therefore the distribution of the modified \hat{z} output by the simulator is both statistically close to the distribution of \hat{z} in the protocol, and consistent with the (N, p, q) values generated by $\mathcal{F}_{\text{RSA Gen}}$. \square

LEMMA A.1. *In Step 5 of the Sampling Phase of $\Pi_{\text{RSA Gen}}$, the distribution of each $p^{(j)}$ value is within statistical distance $(1-\epsilon)\epsilon M_{\text{sample}}/S$ of uniform over $[2^\lambda, 2^{\lambda+\sigma}]$, where $S = 2^{\lambda+\sigma} - 2^\lambda$ is the size of the range and $\epsilon = S/M_{\text{sample}} - \lfloor S/M_{\text{sample}} \rfloor \in [0, 1)$ is the decimal remainder in the division of the range size by M_{sample} .*

PROOF. We can write $S = M_{\text{sample}}(\lfloor S/M_{\text{sample}} \rfloor + \epsilon)$ with $0 \leq \epsilon < 1$. When dividing the range of size S into blocks of size M_{sample} , the last block will not be complete (if M_{sample} does not divide S). When reducing the elements of $[2^\lambda, 2^{\lambda+\sigma}]$ modulo M_{sample} , some residue classes will therefore be present one more time than others: those classes which have representatives in the included portion of the last block. Let X_1 denote the subset of $x \in [2^\lambda, 2^{\lambda+\sigma}]$, whose residue class is more present, and let $X_2 = [2^\lambda, 2^{\lambda+\sigma}] \setminus X_1$ be those elements, whose residue class does not have a representative in the included portion.

Let $x \in [2^\lambda, 2^{\lambda+\sigma}]$, by Euclidean division, x can be uniquely written as $x = aM_{\text{sample}} + b$ with $a \in \mathbb{N}$ and $b \in [0, M_{\text{sample}})$. As $p^{(j)}$ is computed as $p^{(j)} = p'^{(j)} + r^{(j)} \cdot M_{\text{sample}}$ in Step 5 of the sampling phase, this implies:

$$\begin{aligned} \Pr[p^{(j)} = x] &= \Pr[p'^{(j)} = b \wedge r^{(j)} = a] \\ &= \begin{cases} \frac{1}{M_{\text{sample}}} \cdot \frac{1}{\lfloor S/M_{\text{sample}} \rfloor + 1} & x \in X_1, \\ \frac{1}{M_{\text{sample}}} \cdot \frac{1}{\lfloor S/M_{\text{sample}} \rfloor} & x \in X_2, \end{cases} \end{aligned}$$

as $p'^{(j)}$ is uniform in $[0, M_{\text{sample}})$, because of $\mathcal{F}_{\text{ABBWithErrors.Share}}(p')$ and $r^{(j)}$ is uniform subject to the condition that $p^{(j)} \in [2^\lambda, 2^{\lambda+\sigma}]$. Let P denote the above probability distribution. To compute the statistical distance between P and uniform, we compute the size of both X_1 and X_2 , which is

$$\begin{aligned} |X_1| &= \epsilon M_{\text{sample}} \cdot \left(\left\lfloor \frac{S}{M_{\text{sample}}} \right\rfloor + 1 \right) \quad \text{and} \\ |X_2| &= (1-\epsilon) M_{\text{sample}} \cdot \left\lfloor \frac{S}{M_{\text{sample}}} \right\rfloor. \end{aligned}$$

This yields the following distance calculation:

$$\begin{aligned} \Delta(P, U) &= \frac{1}{2} \sum_{x \in [2^\lambda, 2^{\lambda+\sigma}]} |\Pr[P=x] - \Pr[U=x]| \\ &= \frac{1}{2} \left(|X_1| \cdot \left| \Pr[P=x | x \in X_1] - \frac{1}{S} \right| + |X_2| \cdot \left| \Pr[P=x | x \in X_2] - \frac{1}{S} \right| \right) \\ &= \frac{1}{2} \left| \epsilon M_{\text{sample}} \left(\left\lfloor \frac{S}{M_{\text{sample}}} \right\rfloor + 1 \right) \right| \\ &\quad + \frac{1}{2} \left| (1-\epsilon) M_{\text{sample}} \left\lfloor \frac{S}{M_{\text{sample}}} \right\rfloor \right| \\ &= \frac{1}{2} \left(\left| \frac{\epsilon M_{\text{sample}}(\epsilon-1)}{S} \right| + \left| \frac{(1-\epsilon)\epsilon M_{\text{sample}}}{S} \right| \right) \\ &= \frac{(1-\epsilon)\epsilon M_{\text{sample}}}{S}. \end{aligned}$$

\square

B STANDARD FUNCTIONALITIES

We include standard functionalities for MPC for reference.

C UNAUTHENTICATED ARITHMETIC BLACK BOX PROTOCOLS

We include standard protocols for MPC for reference.

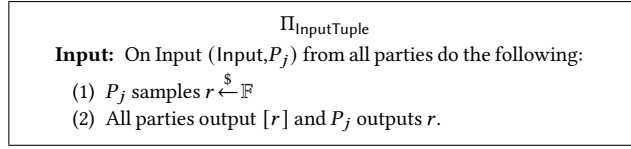


Figure 18: Protocol for passively secure Input Tuples as presented in MASCOT [27].

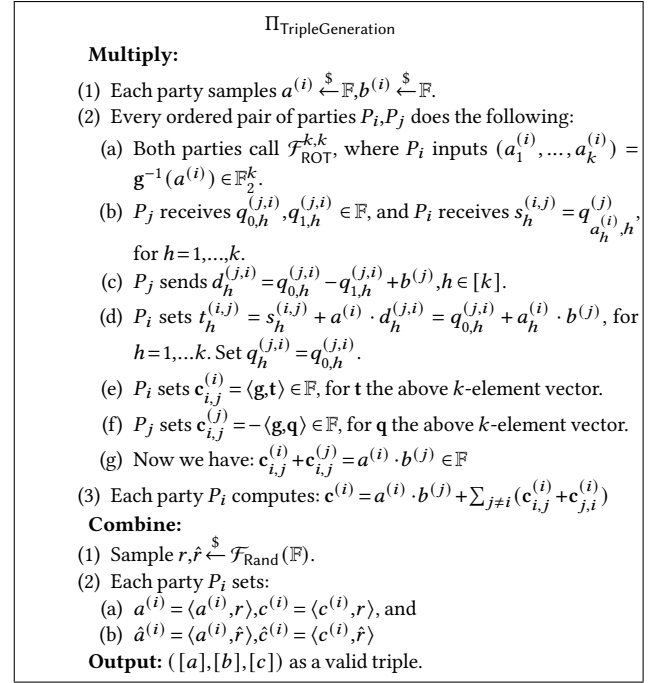


Figure 19: Protocol for Triple Generation adjusted for passive security from MASCOT [27].