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

Cyprien Delpech de Saint Guilhem[1], Eleftheria Makri[1,2], Dragos Rotaru[1,3], and Titouan Tanguy[1]

[1] imec-COSIC, KU Leuven, Belgium
[2] ABRR, Saxion University of Applied Sciences, The Netherlands
[3] Cape Privacy

**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 for generic 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 $\sim 37$.

## 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 [RSA78], 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 [RSA78], 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 were 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 [BF97] 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 [Des94, Rab98, Des98, GRR98, DF90]. 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) [BBBF18, Wes19, Pie19]. 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, Ligero) 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 [BF97]. Boneh and Franklin [BF97] 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 [BF97], 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. [MWB99]. Malkin et al. [MWB99] 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. [MWB99] is a distributed sieving technique, which results in a 10× 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 predeterimined 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. [FMY98] 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. [FMY98], is an unconditionally secure multiplication protocol over the integers. To generate the RSA modulus Frankel et al. [FMY98] 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 [PS98] 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 [Gil99]. 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. [ACS02] perfom a distributed primality test, unlike the protocol of Boneh and Franklin [BF97] 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. [ACS02] and the work of Boneh and Franklin [BF97] are proven secure in the semi-honest, honest majority security model, and require minimally three parties.

Damgård and Mikkelsen [DM10] 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. [FMY98], but malicious security was achieved at the cost of protocol efficiency. The work of Damgård and Mikkelsen [DM10] can be seen as a hybrid of two of its predecessors [BF97, ACS02], 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. [ACS02], 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 [DM10] 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. [HMRT12, HMR$^+$19] 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. [HMRT12, HMR$^+$19] build upon Gilboa's technique [Gil99] achieving two-party actively secure RSA modulus generation à la Boneh and Franklin, but they are the first to adapt also the trial division step of the protocol, achieving therefore a significant efficiency boost. For the multiplication step that follows to compute the candidate biprime, Hazay et al. [HMRT12, HMR$^+$19] carefully use a combination of the Paillier and ElGamal encryption schemes.

In the two party setting, Frederiksen et al. [FLOP18] 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. [HMRT12, HMR$^+$19]. The maliciously secure protocol of Frederiksen et al. [FLOP18] 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. [CCD$^+$20] successfully ovecame the limitations of the proposal of Frederiksen et al. [FLOP18]. Benefiting from the efficiency advantages that a CRT number representation allows, Chen et al. [CCD$^+$20] 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. [FLOP18]. Leveraging the CRT representation, Chen et al. [CCD$^+$20] 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. [CCD$^+$20], named Diogenes [CHI$^+$20], is the most recent result in the area. From a scalability point of view, Diogenes [CHI$^+$20] is the first MPC implementation for a non-trivial task that scales to thousands of parties. To achieve efficiency and scalability, Diogenes [CHI$^+$20] deploys the so-called *coordinator model*, which is a setting consisting of a powerful, honest-but-curious coordinator, and thousands of relatively lightweight computation parties. The RSA modulus generation protocol proposed in Diogenes [CHI$^+$20] is secure against $n-1$ (out of the $n$) malicious parties. Building upon the work of Chen et al. [CCD$^+$20], Diogenes [CHI$^+$20] also deploys the CRT representation, and constructive sampling techniques. To avoid the communication cost that pairwise messages incur, and to exploit the potential of packing and SIMD to the fullest, Diogenes [CHI$^+$20] 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 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.

[*] 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

| Protocol | Security | Dishonest Majority | #Parties | Test | No Leakage |
|---|---|---|---|---|---|
| [BF97] | Passive | × | $n \geq 3$ | biprimality | ✓ |
| [FMY98] | Active | × | $n \geq 3$ | biprimality | ✓ |
| [PS98] | Active | ✓ | $n = 2$ | biprimality | × |
| [Gil99] | Passive | ✓ | $n = 2$ | biprimality | ✓ |
| [ACS02] | Passive | × | $n \geq 3$ | primality | ✓ |
| [DM10] | Active | × | $n = 3^*$ | primality | ✓ |
| [HMRT12]<br>[HMR+19] | Active | ✓ | $n \geq 2$ | biprimality | ✓ |
| [FLOP18] | Active | ✓ | $n = 2$ | biprimality | × |
| [CCD+20] | Active | ✓ | $n \geq 2$ | biprimality | ✓ |
| [CHI+20] | Active$^{**}$ | ✓ | $n \geq 2$ | biprimality | ✓ |
| Ours | Active | ✓ | $n \geq 2$ | biprimality | ✓ |

**Table 1.** Comparison of the related work.

### 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 [Sha79] 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. [AFL+16, FLNW17], should high throughput be the main goal of the MPC implementation.

Following the paradigm of recent work [CCD+20], 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.

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 $\mathsf{sec}$ times, for $\mathsf{sec}$ a statistical security parameter, which means that any cost savings in this part of the protocol impact significantly the overall efficiency. The more computationally and communication intensive consistency check is only performed once, only if 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 $\mathsf{CRT}$ 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. [GRR+16], 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 [KPR18, BCS19], 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 [ST19, HKRW20, DOK+20], 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 [CCD$^+$20] 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 $\mathsf{sec}$ 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 [CCD$^+$20].

### 1.3 Technical Overview

Our main protocol, $\Pi_{\mathsf{RSAGen}}$, 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 introduce with probability $2^{-\mathsf{sec}}$, for $\mathsf{sec}$ a statistical parameter.

Our Sampling phase first deploys a technique similar to the one introduced by Malkin et al. [MWB99], 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 [CCD$^+$20, CHI$^+$20], 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 $\mathcal{F}_{\mathsf{MPC\text{-}CRT}}$.

In the Combine phase of $\Pi_{\mathsf{RSAGen}}$, based on the aforementioned $\mathcal{F}_{\mathsf{MPC\text{-}CRT}}$ 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 $M_{\mathsf{sample}}$, which is the product of $\ell_1$ 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 $2^{-\mathsf{sec}}$ (before proceeding to the ultimate GCD test) we repeat the Jacobi test $\mathsf{sec}$ 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. [GRR$^+$16] to compute the desired Jacobi symbol. If the Jacobi symbol is $\pm 1$, 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, $\Pi_{\mathsf{Rand2k}}$) 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 $\Pi_{\mathsf{ConvInt}}$. By ensuring that indeed no computation wrapped around, we check an equivalent relationship for the exponentiation performed for the 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 [BF97], the latter test introduces a false negative, in the particular case of $N=p\cdot q$, with $p,q$ primes, and $q=1 \bmod p$.

## 2 Preliminaries

### 2.1 Chinese Remainder Theorem - CRT

Following the blueprint of the two most recent works in distributed RSA modulus generation [CCD+20, CHI+20], we deploy in our work the Chinese Remainder Theorem to increase the efficiency of our protocol. We recall here the Chinese Remainder Theorem [KL20].

**Theorem 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 $x \in \{0,...,N-1\}$ to pairs $(x_p,x_q)$ with $x_p \in \{0,...,p-1\}$ and $x_q \in \{0,...,q-1\}$ defined by $f(x)=([x \bmod p],[x \bmod q])$. 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 $p_1,p_2,...,p_\ell$, whose product is $N=\prod_{i=1}^{\ell} p_i$. Then the function $f$ mapping elements $x \in \{0,...,N-1\}$ to tuples $(x_{p_1},...,x_{p_\ell})$ with $x_{p_j} \in \{0,...,p_j-1\}$, is an isomorphism from $\mathbb{Z}_N$ to $\mathbb{Z}_{p_1} \times \cdots \times \mathbb{Z}_{p_\ell}$ and from $\mathbb{Z}_N^*$ to $\mathbb{Z}_{p_1}^* \times \cdots \times \mathbb{Z}_{p_\ell}^*$. We refer to the tuples $(x_{p_1},...,x_{p_\ell})$ as the CRT representation of $x$.

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

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

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

**Fig. 1.** CRT Reconstruction Algorithm.

### 2.2 Notation

We define $M_{\mathsf{sample}}=3\cdot 5 \cdots p_b$ 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 $\ell$ distinct primes of size 128 bits each. To achieve efficient arithmetic over $M_\ell$ we use $\ell$ 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 $M_\ell$.

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

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 $[\![x]\!]$. When the sharings are over the $\mathsf{CRT}$ representation with $\ell$ $\mathsf{CRT}$ components, we denote the sharings as $[x,\ell]$, and $[\![x,\ell]\!]$, respectively, and assume $\ell$ MPC engines operating in parallel, one for each $\mathsf{CRT}$ 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 $\mathcal{F}_{\mathsf{RSAGen}}$ 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 $\mathcal{F}_{\mathsf{RSAGen}}$.

## 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 $\Pi_{\mathsf{RSAGen}}$ protocol, which in turn realizes the $\mathcal{F}_{\mathsf{RSAGen}}$ functionality.

The first functionality that our protocol makes use of, is the $\mathcal{F}_{\mathsf{ABBWithErrors}}$. This is used in the sampling phase of $\Pi_{\mathsf{RSAGen}}$, 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 $\Pi_{\mathsf{ABBWithErrors}}$ protocol, which in turn is constructed in the $\mathcal{F}_{\mathsf{ABBWithErrors-Prep}}$-hybrid model (realized by $\Pi_{\mathsf{ABBWithErrors-Prep}}$). The reader can think of this functionality as the standard MPC arithmertic 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 $\Pi_{\mathsf{ABBWithErrors}}$ in Section 3.2.

Then, the rest of the sampling phase, as well as the combining phase of $\Pi_{\mathsf{RSAGen}}$ uses the $\mathcal{F}_{\mathsf{MPC\text{-}CRT}}$ functionality, realized by the $\Pi_{\mathsf{MPC\text{-}CRT}}$ protocol, which is in turn designed in the standard $\mathcal{F}_{\mathsf{MPC}}$-hybrid model. For completeness, we present the $\mathcal{F}_{\mathsf{MPC}}$ functionality in Fig. 16, 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 $\mathsf{CRT}$ representation of the inputs, designing therefore the $\mathcal{F}_{\mathsf{MPC\text{-}CRT}}$ functionality.

The Jacobi test phase of $\Pi_{\mathsf{RSAGen}}$ makes use of the standard broadcast, and randomness sampling functionalities, which are presented for completeness in Appendix B, Fig. 17, and Fig. 18, respectively. The consistency check that follows the Jacobi test of $\Pi_{\mathsf{RSAGen}}$ requires two additional functionalities. To support these two additional functionalities, we augment the $\mathcal{F}_{\mathsf{MPC\text{-}CRT}}$ functionality with two additional commands, and integrate them into the $\Pi_{\mathsf{AdvMPC-CRT}}$ protocol, realizing the corresponding $\mathcal{F}_{\mathsf{AdvMPC-CRT}}$ 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 $\mathsf{CRT}$ representation. This construction is presented in Section 3.4 and the protocol that realizes it works in the $\mathcal{F}_{\mathsf{maBits}}$-hybrid model. The latter functionality facilitates the generation of multiply authenticated random bits [RST$^{+}$19]. 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 $\mathsf{CRT}$ representation to an additive sharing of the $\mathsf{CRT}$ reconstructed value over the integers. We explain how to achieve this in Section 3.4.

## 3.2 Unauthenticated Arithmetic Black Box Functionality

$\mathcal{F}_{\mathsf{ABBWithErrors}}$ (Fig. 3) is the functionality implementing an unauthenticated arithmetic black box MPC. Our $\Pi_{\mathsf{RSAGen}}$ 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 $\Pi_{\mathsf{RSAGen}}$ are actively secure.

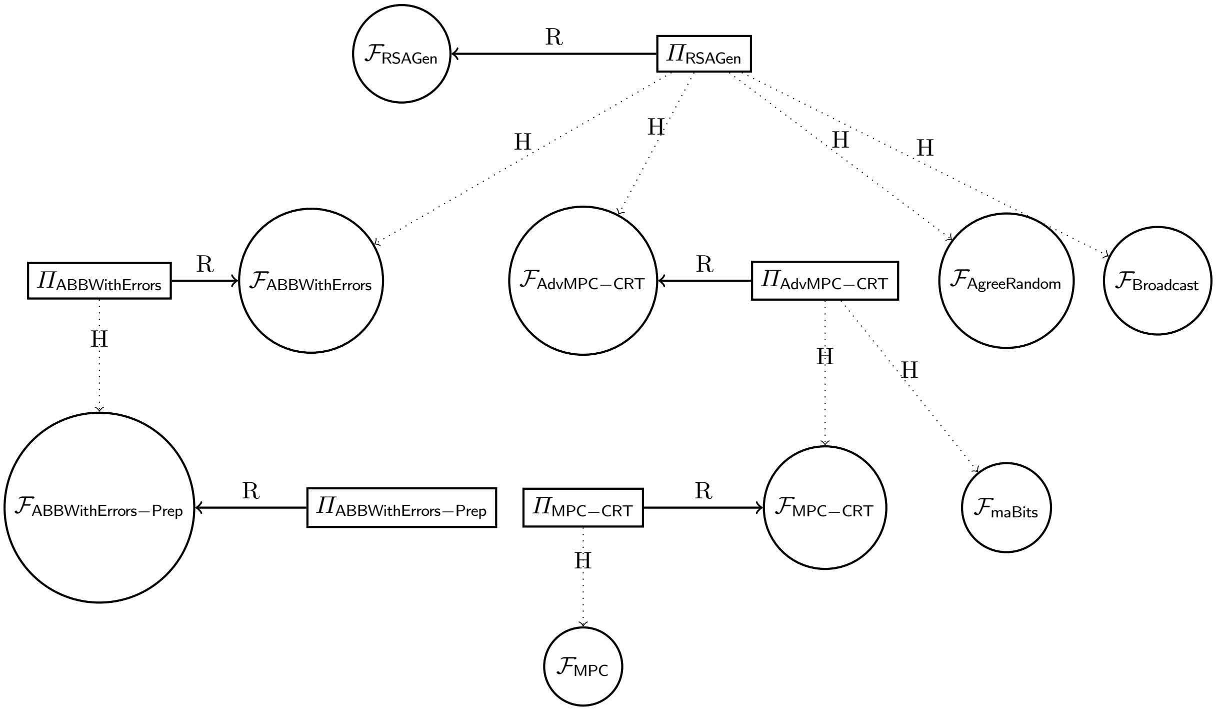
$\mathcal{F}_{\mathsf{RSAGen}}$
R
$\Pi_{\mathsf{RSAGen}}$
H
H
H
H
$\Pi_{\mathsf{ABBWithErrors}}$
R
$\mathcal{F}_{\mathsf{ABBWithErrors}}$
$\mathcal{F}_{\mathsf{AdvMPC-CRT}}$
R
$\Pi_{\mathsf{AdvMPC-CRT}}$
$\mathcal{F}_{\mathsf{AgreeRandom}}$
$\mathcal{F}_{\mathsf{Broadcast}}$
H
H
H
$\mathcal{F}_{\mathsf{ABBWithErrors-Prep}}$
R
$\Pi_{\mathsf{ABBWithErrors-Prep}}$
$\Pi_{\mathsf{MPC-CRT}}$
R
$\mathcal{F}_{\mathsf{MPC-CRT}}$
$\mathcal{F}_{\mathsf{maBits}}$
H
$\mathcal{F}_{\mathsf{MPC}}$

**Fig. 2.** Functionality dependencies for RSA modulus generation.

For completeness, we detail the protocol realizing the unauthenticated ABB functionality, $\Pi_{\mathsf{ABBWithErrors}}$, in Appendix A, Fig. 13. The $\Pi_{\mathsf{ABBWithErrors}}$ protocol implements the online phase of the unauthenticated arithmetic black box, and it works in the $\mathcal{F}_{\mathsf{ABBWithErrors-Prep}}$-hybrid model. This functionality, realized by $\Pi_{\mathsf{ABBWithErrors-Prep}}$, 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 $\Pi_{\mathsf{InputTuple}}$ (Fig. 14), while the protocol for the preprocessing of triples is listed in $\Pi_{\mathsf{TripleGeneration}}$ (Fig. 15), in Appendix A. Note that the $\Pi_{\mathsf{TripleGeneration}}$ protocol makes use of the standard $\mathcal{F}_{\mathsf{Rand}}$ functionality, which is presented for completeness in Fig. 19, Appendix B.

For simplicity and clarity of presentation we present here (and in Appendices A 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 $\mathsf{CRT}$ representation of the sharings and inputs, meaning that we would need multiple MPC engines operating in parallel for each $\mathsf{CRT}$ component. We detail how to achieve the standard actively secure MPC functionality over $\mathsf{CRT}$ components in Section 3.3. The $\mathcal{F}_{\mathsf{ABBWithErrors}}$ functionality can be also implemented over $\mathsf{CRT}$ components in the same manner. We have assumed the aforemenioned implementation of the $\mathcal{F}_{\mathsf{ABBWithErrors}}$ functionality for the efficiency analysis of our protocol.

**Functionality $\mathcal{F}_{\mathsf{ABBWithErrors}}$**

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

**Fig. 3.** Arithmetic Black Box Functionality with Errors.

### 3.3 MPC on CRT Components

In this subsection we describe a new functionality and its associated protocol to perform secure multiparty computation over a big composite modulus, by relying on the Chinese Remainder Theorem. The functionality $\mathcal{F}_{\mathsf{MPC-CRT}}$ (Fig. 4) essentially implements the standard MPC functionality, but on sharings in their $\mathsf{CRT}$ representation. To accommodate computations on this type of sharings, we deploy $\ell$ MPC engines, for $\ell$ the maximum possible number of $\mathsf{CRT}$ components in the representation, as shown in $\varPi_{\mathsf{MPC-CRT}}$ (Fig. 5). Each of these MPC engines operates over one of the prime moduli of the $\mathsf{CRT}$ representation, and each of these $\ell$ prime moduli is 128-bits long. All $\ell$ 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 $\varPi_{\mathsf{RSAGen}}$ protocol.

**Functionality $\mathcal{F}_{\mathsf{MPC-CRT}}$**

Let $[x,\ell]$ denote the identifiers for the $\ell$ components of the $\mathsf{CRT}$ representation a value $x$ stored in the functionality.
Let $A\subset\{1,...,n\}$ denote the index set of the corrupted parties.
**Init:** Receive $p_1,...,p_\ell$ primes from all parties, store them and compute $M_\ell=\varPi_{i=1}^{\ell}p_i$.
**Input:** Receive a tuple $(x,\ell')\in\mathbb{Z}_{M_{\ell'}}$ with $\ell'\leq\ell$ from some party and store $([x,\ell'])$.
**Add**$([x,\ell'],[y,\ell'])$**:** Check if $(x,\ell')$ and $(y,\ell')$ exist in memory. If so compute $z=x+y \bmod M_{\ell'}$ and store $([z,\ell'])$.
**Mult**$([x,\ell'],[y,\ell'])$**:** Check if $(x,\ell')$ and $(y,\ell')$ exist in memory. If so, compute $z=x\cdot y \bmod M_{\ell'}$ and store $([z,\ell'])$.
**Open**$([x,\ell'])$**:** Check if $(x,\ell')$ exists in memory.If so send the value $x$ to all parties.
**OpenTo**$([x,\ell'],j)$**:** Check if $(x,\ell')$ exists in memory. If so send the CRT represented values $(x,\ell')$ to party $P_j$.

**Fig. 4.** MPC over CRT Functionality

$\varPi_{\mathsf{MPC-CRT}}$

**Init**$(\ell)$**:** To initialize $\ell$ CRT MPC engines, parties call $\mathcal{F}_{\mathsf{MPC}}.\mathsf{Init}(\mathbb{F}_{m_i})$ $\forall$ $\ell$ primes $[m_1,m_2,...,m_\ell]$.
**Input**$(x,\ell')$**:** To provide an input $x\in\mathbb{Z}_{M_{\ell'}},\ell'\leq\ell$, any party calls $\mathcal{F}_{\mathsf{MPC}}.\mathsf{Input}(x \bmod m_i)$ $\forall$ $\ell'$ primes $[m_1,m_2,...,m_{\ell'}]$, and defines $[\![x,\ell']\!]$ as $((x,\ell')^{(1)},(x,\ell')^{(2)},...,(x,\ell')^{(n)})$, where each $(x,\ell')^{(j)}$ represents the $\ell'$ $\mathsf{CRT}$ shares that each player $j$ obtains.
**Add**$([\![z,\ell']\!],[\![x,\ell']\!],[\![y,\ell']\!])$**:** To add two shared values $[\![x,\ell']\!],[\![y,\ell']\!]$ parties call $\mathcal{F}_{\mathsf{MPC}}.\mathsf{Add}([\![z,\ell_i]\!],[\![x,\ell_i]\!],[\![y,\ell_i]\!])$ $\forall$ $\ell'$ MPC engines operating over $[m_1,m_2,...,m_{\ell'}]$, and set $[\![z,\ell']\!]\leftarrow[\![x,\ell']\!]+[\![y,\ell']\!]$.
**Mult**$([\![z,\ell']\!],[\![x,\ell']\!],[\![y,\ell']\!])$**:** To multiply two shared values $[\![x,\ell']\!],[\![y,\ell']\!]$ parties call $\mathcal{F}_{\mathsf{MPC}}.\mathsf{Mult}([\![z,\ell_i]\!],[\![x,\ell_i]\!],[\![y,\ell_i]\!])$ $\forall$ $\ell'$ MPC engines operating over $[m_1,m_2,...,m_{\ell'}]$, and set $[\![z,\ell']\!]\leftarrow[\![x,\ell']\!]\cdot[\![y,\ell']\!]$.
**Open**$([\![x,\ell']\!])$**:** To open a shared value $[\![x,\ell']\!]$ all parties call $\mathcal{F}_{\mathsf{MPC}}.\mathsf{Open}([\![x,\ell_i]\!])$ $\forall$ $\ell'$ MPC engines operating over $[m_1,m_2,...,m_{\ell'}]$, and receive $(x,\ell')$.
**OpenTo**$([\![x,\ell']\!],j)$**:** To open a shared value $[\![x,\ell']\!]$ party $P_j$ calls $\mathcal{F}_{\mathsf{MPC}}.\mathsf{OpenTo}([\![x,\ell_i]\!],j)$ $\forall$ $\ell'$ MPC engines operating over $[m_1,m_2,...,m_{\ell'}]$, and receives $(x,\ell')$.

**Fig. 5.** Protocol for arithmetic MPC over $\mathsf{CRT}$moduli.

### 3.4 Advanced MPC CRT

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

**Functionality $\mathcal{F}_{\mathsf{AdvMPC-CRT}}$**

This functionality reproduces all the commands of $\mathcal{F}_{\mathsf{MPC-CRT}}$ and extends it with:
**Rand2k**($\ell'$,$k$)**:** Sample $r\overset{\$}{\leftarrow}\mathbb{Z}_{2^k}$ and store ($[r,\ell']$), for $\ell'\leq\ell$.
**ConvInt**($[x,\ell']$)**:** Check if $(x,\ell')$ exists in memory. If so, uniformly sample shares $x_{\mathsf{Int}}^{(i)}\in\mathbb{Z}$ for each party $P_i$ s.t. $\sum_{i=1}^{n}x_{\mathsf{Int}}^{(i)}=x$, and send them to the corresponding parties. (Note: the sum is taken in $\mathbb{Z}$).
**LevelUp**($[p,\ell]$,$[q,\ell]$,$\ell'$)**:** Receive $p_{\ell+1},...,p_{\ell'}$ from all parties, store them and compute $M_{\ell'}=\prod_{i=1}^{\ell'}p_i$. Store ($[p,\ell']$) and ($[q,\ell']$).

**Fig. 6.** Advanced MPC over CRT Functionality

**Bounded Randomness in Shared CRT Representation.** The $\mathsf{Rand2k}$ command allows us to sample a random $\mathsf{CRT}$ 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 intial (smaller) $\mathsf{CRT}$ representation. The protocol implementing the $\mathsf{Rand2k}$ command is listed in $\Pi_{\mathsf{Rand2k}}$ (Fig. 8), and it is designed in the $\mathcal{F}_{\mathsf{maBits}}$-hybrid model. The $\mathcal{F}_{\mathsf{maBits}}$ functionality itself, presented in Fig. 7, is a slightly different version of the one presented by Rotaru et al. [RST$^+$19]. In our case, we modify our functionality so that it outputs the integer sharing of the bit, which was discarded in the original $\mathcal{F}_{\mathsf{maBits}}$ functionality. We need this integer sharing later, in the $\Pi_{\mathsf{ConvInt}}$ protocol.

**Functionality $\mathcal{F}_{\mathsf{maBits}}$**

1. For $i=1,...,m+\gamma\cdot\mathsf{sec}$ the functionality calls $\mathcal{F}_{\mathsf{MPC}}.\mathsf{GenBit}()$ so as to store a bit $b_i$.
2. The bits $b_i$ are retrieved from $\mathcal{F}_{\mathsf{MPC}}$ and are enterred into the $\mathcal{F}_{\mathsf{MPC\text{-}CRT}}$ functionality.
3. The functionality samples a sharing of $b_i$ in $\mathbb{Z}_p$ and send its share $b_i^{(j)}$ to every party $P_j$.
   It also publicly outputs $k_i=\lfloor\frac{\sum b_i^{(j)}}{p}\rfloor$.
4. The functionality waits for a message $\mathsf{Abort}$ or $\mathsf{Ok}$ from the adversary. If the message is $\mathsf{Ok}$ then it continues.
5. The functionality waits for a proper subset $\mathcal{S}$ of $\{1,...,m\}$ from the adversary and returns the bits $b_j$ for $j\in\mathcal{S}$ to the adversary. (We note that this subset can be adaptively chosen, but to simplify the exposition here, we assume it is presented to the functionality in one go.)

**Fig. 7.** The ideal functionality for generating random bits

$\Pi_{\mathsf{Rand2k}}$

**Rand2k:** On input $(\ell',k)$, to generate a random $\mathsf{CRT}$ sharing $\llbracket r,\ell' \rrbracket$ with $r<2^k$, parties do the following:

1. All parties call $\mathcal{F}_{\mathsf{maBits}}$ to generate $k$ random bits $\{\llbracket b_i,\ell' \rrbracket\}_{i\in[k]}$ shared across all MPC engines, and receive $k_i$ which gives an integer sharing of $b_i$ w.r.t. the first $\mathsf{CRT}$ component of $\llbracket b_i,\ell' \rrbracket$.
2. All parties compute $\llbracket r,\ell' \rrbracket=\sum_{i\in[k]}2^i\cdot\llbracket b_i,\ell' \rrbracket$.
3. Output $\llbracket r,\ell' \rrbracket$ and the integer sharing $[r_{\mathsf{Int}}]$.

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

**Converting a CRT Sharing to an Integer Sharing.** Our main protocol requires a command which converts a $\mathsf{CRT}$ sharing to an integer sharing of the same underlying secret. This is necessary during the Jacobi test of $\Pi_{\mathsf{RSAGen}}$, 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 $\mathsf{ConvInt}$ command with the protocol $\Pi_{\mathsf{ConvInt}}$, listed in Fig. 9. This protocol allows the parties to obtain an unauthenticated integer sharing of the $\mathsf{CRT}$ sharings they already hold, without leaking any information about the underlying secret value.

$\Pi_{\mathsf{ConvInt}}$

Let $B\in\mathbb{N}$ be an upper bound for the bit-length of the input; i.e. for any input $x$, we assume $x<2^B$. We select $\ell_{\mathsf{Jac}}$, such that $M_{\ell_{\mathsf{Jac}}}=\prod_{i=1}^{\ell_{\mathsf{Jac}}}m_i$ is the minimal moduli product bigger than $2^{B+\mathsf{sec}+1}$

**ConvInt:** On input $\llbracket x,\ell_{\mathsf{Jac}} \rrbracket$, to convert the CRT sharing $\llbracket x,\ell_{\mathsf{Jac}} \rrbracket$ to an integer sharing $[x_{\mathsf{Int}}]$ that reconstructs to the same underlying secret, parties do the following:

1. Parties call $\mathsf{Rand2k}$ with input $(\ell_{\mathsf{Jac}},B+\mathsf{sec})$ to get a CRT sharing $\llbracket r,\ell_{\mathsf{Jac}} \rrbracket$ and an integer sharing $[r_{\mathsf{Int}}]$ of a random value $r$, with $r<2^{B+\mathsf{sec}}$.
2. Parties call $(t,\ell_{\mathsf{Jac}})=\mathcal{F}_{\mathsf{MPC\text{-}CRT}}.\mathsf{Open}(\llbracket x,\ell_{\mathsf{Jac}} \rrbracket+\llbracket r,\ell_{\mathsf{Jac}} \rrbracket)$ and do the local CRT reconstruction $t=\mathsf{CRTrec}(t,\ell_{\mathsf{Jac}})$.
3. To obtain an integer sharing of $x$ parties locally compute $x_{\mathsf{Int}}^{(i)}=t-r^{(i)}$.
4. Parties store $x_{\mathsf{Int}}^{(i)}$ as their integer share of $x$.

**Fig. 9.** Protocol for converting a bounded CRT sharing to an integer sharing.

The execution of the protocol $\Pi_{\mathsf{ConvInt}}$ does not leak any information about the secret. Indeed, the only opened value in the protocol is $\llbracket x,\ell_{\mathsf{Jac}} \rrbracket+\llbracket r,\ell_{\mathsf{Jac}} \rrbracket$ with $r\overset{\$}{\leftarrow}[2^{B+\mathsf{sec}}]$ and $x<2^B$. Therefore, by Theorem 2, we have that the distance between the distribution of this opened value and the uniform distribution is upper bounded by $2^{-\mathsf{sec}}$.

This protocol only produces an *unauthenticated* integer sharing, but the consistency of the shares is checked later in the $\Pi_{\mathsf{RSAGen}}$ 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.

**Theorem 2 ( [ST06, Appendix A]).** *Let $M$ and $K$ be positive integers, where $M\leq K$. Let the random variable $X$ take values from $\{0,...,M-1\}$ and let the random variables $U$ be uniform on $\{0,...,K-1\}$. Then $\Delta(U,X+U)\leq(M-1)/K$ is an upper bound for the distance between the two distributions.*

**Extending the CRT Representation.** The $\mathsf{LevelUp}$ command extends the $\mathsf{CRT}$ representation of $[p,\ell]$ and $[q,\ell]$, allowing us to compute over $M_{\ell'}>M_\ell$ for these two values. In our $\Pi_{\mathsf{RSAGen}}$ protocol, we use $\mathsf{LevelUp}$ command whenever a new operation on $p$ and $q$ could overflow the current $\mathsf{CRT}$ 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 have passed the biprimality test, and for which the product $N$ is publicly known. We use both of these properties in the $\Pi_{\mathsf{LevelUp}}$ protocol (Fig. 10), which implements the $\mathsf{LevelUp}$ command.

$\Pi_{\mathsf{LevelUp}}$

For the execution of this protocol, we assume that the parties have access to an integer sharing of the values they wish to extend, and that the product of the values to be reshared has passed the biprimality test.

**LevelUp:** On input $(\llbracket p,\ell\rrbracket,\llbracket q,\ell\rrbracket,\ell')$, with $\ell'>\ell$, publicly known $N=p\cdot q$ parties do the following:

1. Each party $P_j$ retrieves its integer sharings $p^{(j)}$ and $q^{(j)}$ of $\llbracket p,\ell\rrbracket$ and $\llbracket q,\ell\rrbracket$.
2. Each party $P_j$ calls $\mathcal{F}_{\mathsf{MPC}}.\mathsf{Input}(p^{(j)} \bmod m_i)$ and $\mathcal{F}_{\mathsf{MPC}}.\mathsf{Input}(q^{(j)} \bmod m_i)$ for $i\in\{\ell+1,\ldots,\ell'\}$.
3. All parties call $\mathcal{F}_{\mathsf{MPC-CRT}}.\mathsf{Add}$ to obtain $\llbracket p,\ell'\rrbracket=\sum_{j=1}^{n}\llbracket p^{(j)},\ell'\rrbracket$ and $\llbracket q,\ell'\rrbracket=\sum_{j=1}^{n}\llbracket q^{(j)},\ell'\rrbracket$.
4. All parties call $\llbracket N,\ell'\rrbracket=\mathcal{F}_{\mathsf{MPC-CRT}}.\mathsf{Mult}(\llbracket p,\ell'\rrbracket,\llbracket q,\ell'\rrbracket)$ and then call $\mathcal{F}_{\mathsf{MPC-CRT}}.\mathsf{Open}(\llbracket N,\ell'\rrbracket)$, abort if the result is not equal to the publicly known $N$.

**Fig. 10.** Protocol for extending the $\mathsf{CRT}$ representation of $\llbracket p,\ell\rrbracket$ and $\llbracket q,\ell\rrbracket$.

The protocol ensures that the new $\llbracket p,\ell'\rrbracket$ and $\llbracket q,\ell'\rrbracket$ are sharings of the same value as $\llbracket p,\ell\rrbracket$ and $\llbracket q,\ell\rrbracket$. Indeed, checking that both multiply to $N$ acts as a MAC check. To be successful, a cheating adversary would have to introduce additive errors on $\llbracket p,\ell'\rrbracket=\llbracket p,\ell\rrbracket+\Delta_p$ and $\llbracket q,\ell'\rrbracket=\llbracket q,\ell\rrbracket+\Delta_q$, such that $\Delta_p\cdot q+\Delta_q\cdot p+\Delta_q\cdot\Delta_p=0$; that is an adversary would have to guess either $p$ or $q$ from $N$. Because $N$ is an RSA modulus, we assumed that factoring $N$ is a hard problem, thus making $\Pi_{\mathsf{LevelUp}}$ secure.

## 4 Distributed Generation of an RSA Biprime

Our ideal functionality $\mathcal{F}_{\mathsf{RSAGen}}$, 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 $M_{\mathsf{sample}}$, 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 $M_{\mathsf{sample}}$. The third step follows the blueprint of Boneh and Franklin [BF97], checking biprimality in a way similar to Miller-Rabin primality testing, and returing $\mathsf{Abort}$, if the product computed is not a biprime. The fourth step serves as a constistency 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 $\Pi_{\mathsf{RSAGen}}$ (Fig. 12), realizing the $\mathcal{F}_{\mathsf{RSAGen}}$ functionality. Each party $P_j$ samples a multiplicative share $\hat{p}^{(j)}$, such that $\mathsf{gcd}(M_{\mathsf{sample}},\hat{p}^{(j)})=1$. The goal of the Sampling phase is to convert the multiplicative sharing $\hat{p}=\hat{p}^{(1)}\cdots\hat{p}^{(n)}$ over the integers, into an additive sharing $p'=p'^{(1)}+\cdots+p'^{(n)}$ over $\mathbb{Z}/(M_{\mathsf{sample}}\mathbb{Z})$, such that $p'=\hat{p} \bmod M_{\mathsf{sample}}=p'^{(1)}+\cdots+p'^{(n)}=\hat{p}^{(1)}\cdots\hat{p}^{(n)} \bmod M_{\mathsf{sample}}$. 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 $\hat{p}^{(j)}=\hat{p}_1^{(j)}+\cdots+\hat{p}_n^{(j)} \bmod M_{\mathsf{sample}}$. After the multiplication is done over $\mathbb{Z}/(M_{\mathsf{sample}}\mathbb{Z})$, parties set their local share as $p^{(j)}=p'^{(j)}+r^{(j)}\cdot M_{\mathsf{sample}}$ and use this in the $\mathsf{CRT}$ Input procedure, over $\ell_c$ $\mathsf{CRT}$ components, and thus $\ell_c$ 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 $\ell_c\mathsf{CRT}$ components. Then, the candidate biprime is computed, using an actively secure multiplication over the $\mathsf{CRT}$ 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 $\mathsf{CRT}$ 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.

**Functionality $\mathcal{F}_{\mathsf{RSAGen}}$**

For $\lambda$ the bit-length of each of the candidate primes $p$ and $q$ we aim to sample, and $\sigma$ a statistical security parameter, let $\ell_1$ be the number of primes, the product of which ($M_{\mathsf{sample}}$) serves as the space over which we sample the candidate primes $p$ and $q$, such that $\prod_{i=1}^{\ell_1} m_i = M_{\mathsf{sample}} > 2^{\lambda+\sigma}$. In addition, let $\ell_{\mathsf{c}}$ be the number of primes, the product of which ($M_{\ell_{\mathsf{c}}}$) serves as the space over which we compute without overflow the product of the candidate primes $p$ and $q$, such that $\prod_{i=1}^{\ell_{\mathsf{c}}} m_i = M_{\ell_{\mathsf{c}}} > 2^{2\cdot(\lambda+\sigma)}$.

1. On receiving Sample from all parties, First query $\mathcal{S}$ for the values $m_p = p \mod M_{\mathsf{sample}}$ and $m_q = q \mod M_{\mathsf{sample}}$ together with the shares $p^{(j)}, q^{(j)}$ of corrupt parties $P_j \in \mathcal{C}$. Then, uniformly sample $p^{(j)}, q^{(j)}$ and send them to honest parties $P_j \in \mathcal{H}$ with the condition that $\sum_j p^{(j)} \mod M_{\mathsf{sample}} = m_p$ and $\sum_j q^{(j)} \mod M_{\mathsf{sample}} = m_q$.
2. On receiving Combine from all parties, send $N_i = (p\cdot q) \bmod m_i$ for all $i \in [\ell_{\mathsf{c}}]$ to all parties. If $\mathsf{gcd}(N, M_{\mathsf{sample}}) \neq 1$ send $\mathsf{Abort}_{\mathsf{GCD}}$ to all parties; if $N \notin [2^{2\lambda}, 2^{2\cdot(\lambda+\sigma)}]$ send $\mathsf{Abort}_{\mathsf{OutOfRange}}$ to all parties.
3. On receiving Jacobi($\gamma$) from all parties, First, compute $y = \gamma^{(N-p-q+1)/4} \mod N$ and send $y$ to $\mathcal{S}$. Then, receive $y'$ from $\mathcal{S}$ and send $y'$ to all parties. Finally, if $y' \neq \pm 1$ then send Abort to all parties.
4. On receiving Consistency Check from all parties, if $y'$ received during the Jacobi command was not equal to $y$, then send Abort to all parties.
5. On receiving GCD Test from all parties, compute $b = \mathsf{gcd}(N, (p+q-1))$. If $b = 1$, send ($b$,Biprime) to all parties, otherwise send ($b$,Non Biprime).

**Fig. 11.** RSA Modulus Generation Functionality

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 $\ell_c$ components of the combine step, to $\ell_{\mathsf{Jac}}$ 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 $\ell_{\mathsf{Jac}}$ 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 $\ell_{\mathsf{Jac}}$ components to an integer sharing by calling the ConvInt command. The integer sharing is used to exponentiate the public value $\gamma$ 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 $[\![p+q-1]\!]$, and if their GCD equals 1, the test passes. Let $Q_{\mathsf{gcd}} > V \cdot N$, where $V = 2^{3\lambda+4\sigma}$. The goal is to output the product $a\cdot(p+q-1) + v\cdot N \bmod Q_{\mathsf{gcd}}$, 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, $M_{\ell_{\mathsf{gcd}}}$ is computed, so that $v\cdot N$ fits $Q_{\mathsf{gcd}}$, which makes $M_{\ell_{\mathsf{gcd}}} 5\lambda+6\sigma$ 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 $\ell_{\mathsf{gcd}}$ 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 reconstuction 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.

$\Pi_{\mathsf{RSAGen}}$

**Sampling phase.** All the steps below are done in parallel for $p$ and $q$.

1. Each party $P_j$ samples a multiplicative share $\hat{p}^{(j)}$, such that $\mathsf{gcd}(M_{\mathsf{sample}},\hat{p}^{(j)})=1$.
2. Each party $P_j$ calls $\mathcal{F}_{\mathsf{ABBWithErrors}}.\mathsf{Input}(\hat{p}^{(j)})$.
3. The parties call $\mathcal{F}_{\mathsf{ABBWithErrors}}.\mathsf{Mult}(p'',\hat{p}^{(1)},...,\hat{p}^{(n)})$.
4. Parties call $\mathcal{F}_{\mathsf{ABBWithErrors}}.\mathsf{Share}(p')$, such that $P_j$ receives the residues of $p''^{(j)}$ for all primes in $M_{\mathsf{sample}}$.
5. Parties reconstruct $p'^{(j)}=\mathsf{CRTRec}([p'^{(j)},3\cdot\delta_{j,0}],[M_{\mathsf{sample}},4])$ where $\delta_{j,0}$ is the Kronecker delta.
6. Each party $P_j$ samples $r^{(j)}$, and computes $p^{(j)}=p'^{(j)}+r^{(j)}\cdot 4\cdot M_{\mathsf{sample}}$, such that $p^{(j)}\in[2^{\lambda},2^{\lambda+\sigma}]$, for $\sigma$ a statistical security parameter.
7. Each party $P_j$ calls $\mathcal{F}_{\mathsf{MPC\text{-}CRT}}.\mathsf{Input}(p^{(j)},\ell_{\mathsf{c}})$.

**Combine**

1. Parties call $\mathcal{F}_{\mathsf{MPC\text{-}CRT}}.\mathsf{Sum}([\![p,\ell_{\mathsf{c}}]\!],[\![p^{(1)},\ell_{\mathsf{c}}]\!],...,[\![p^{(n)},\ell_{\mathsf{c}}]\!])$ and $\mathcal{F}_{\mathsf{MPC\text{-}CRT}}.\mathsf{Sum}([\![q,\ell_{\mathsf{c}}]\!],[\![q^{(1)},\ell_{\mathsf{c}}]\!],...,[\![q^{(n)},\ell_{\mathsf{c}}]\!])$.
2. Parties call $\mathcal{F}_{\mathsf{MPC\text{-}CRT}}.\mathsf{Mult}([\![N,\ell_{\mathsf{c}}]\!],[\![p,\ell_{\mathsf{c}}]\!],[\![q,\ell_{\mathsf{c}}]\!])$.
3. Parties call $\mathcal{F}_{\mathsf{MPC\text{-}CRT}}.\mathsf{Open}([\![N,\ell_{\mathsf{c}}]\!])$.
4. Each party locally reconstructs $N=\mathsf{CRTrec}(N,\ell_{\mathsf{c}})$, checks that $N\in[2^{2\lambda},2^{2(\lambda+\sigma)}]$, and $GCD(M_{\mathsf{sample}},N)=1$, abort if false.

**Jacobi test** This is executed $\mathsf{sec}$ times (Grassi et al. fashion but carefully so that adding shares in the exponents is done over the integers).

1. Parties call $\mathcal{F}_{\mathsf{AgreeRandom}}$ to sample a public $\gamma\in\mathbb{Z}_N$. Repeat until Jacobi symbol $(\frac{\gamma}{N})=1$.
2. Using their integer shares of $p$ and $q$, $P_1$ computes $y^{(1)}=\gamma^{(N-p^{(1)}-q^{(1)}+1)/4} \bmod N$ and calls $\mathcal{F}_{\mathsf{Broadcast}}(y^{(1)})$, and each party $P_j,j\neq 1$ computes $y^{(j)}=\gamma^{(-p^{(j)}-q^{(j)})/4} \bmod N$ and calls $\mathcal{F}_{\mathsf{Broadcast}}(y^{(j)})$.
3. All parties compute $y=\prod_{j=1}^{n}y^{(j)}$.
4. If $y\neq\pm1$ Abort.

**Consistency Check**

1. Parties call $\mathcal{F}_{\mathsf{AdvMPC-CRT}}.\mathsf{LevelUp}$ with input $(\ell_{\mathsf{Jac}}, [\![p,\ell_c]\!], [\![q,\ell_c]\!])$ and publicly known $N=pq$, and receive $([\![p,\ell_{\mathsf{Jac}}]\!],[\![q,\ell_{\mathsf{Jac}}]\!])$
2. Parties call $\mathcal{F}_{\mathsf{AdvMPC-CRT}}.\mathsf{Rand2k}$ with input $(\ell_{\mathsf{Jac}},\mathsf{sec})$ to get a CRT sharing $[\![x,\ell_{\mathsf{Jac}}]\!]$ of a random value $x$, bounded by $2^{\mathsf{sec}}$.
3. All parties call $\mathcal{F}_{\mathsf{MPC\text{-}CRT}}.\mathsf{Mult}([\![t,\ell_{\mathsf{Jac}}]\!],[\![x,\ell_{\mathsf{Jac}}]\!],[\![((N-p-q+1)/4),\ell_{\mathsf{Jac}}]\!])$, where the multiplication result is actually bounded by $M_{\ell_{\mathsf{c}}}$ and CRT shared in $M_{\ell_{\mathsf{Jac}}}$.
4. Parties call $\mathcal{F}_{\mathsf{AdvMPC-CRT}}.\mathsf{ConvInt}([\![t,\ell_{\mathsf{Jac}}]\!])$ to obtain an additive sharing of $t$ over the integers, denoted as $[t]_{\mathsf{Int}}$.
5. Each party calls $\mathcal{F}_{\mathsf{Broadcast}}(\gamma^{t^{(j)}_{\mathsf{Int}}})$.
6. All parties call $\mathcal{F}_{\mathsf{MPC\text{-}CRT}}.\mathsf{Open}([\![x,\ell_{\mathsf{Jac}}]\!])$, and compute $x=\mathsf{CRTrec}(x,\ell_{\mathsf{Jac}})$.
7. All parties locally check that $\prod_{j=1}^{n}\gamma^{t^{(j)}_{\mathsf{Int}}}=y^{x}$. Abort if equality fails.

**GCD test**

1. Parties call $\mathcal{F}_{\mathsf{AdvMPC-CRT}}.\mathsf{LevelUp}$ with input $(\ell_{\mathsf{gcd}}, [\![p,\ell_c]\!], [\![q,\ell_c]\!])$ and publicly known $N=pq$, and receive $([\![p,\ell_{\mathsf{gcd}}]\!],[\![q,\ell_{\mathsf{gcd}}]\!])$
2. Parties call $\mathcal{F}_{\mathsf{AdvMPC-CRT}}.\mathsf{Rand2k}$ with input $(\ell_{\mathsf{gcd}},2\lambda+2\sigma)$ to get a CRT sharing $[\![a,\ell_{\mathsf{gcd}}]\!]$ of a random value $a$, bounded by $2^{2\lambda+2\sigma}$.
3. All parties call $\mathcal{F}_{\mathsf{MPC\text{-}CRT}}.\mathsf{Mult}([\![z,\ell_{\mathsf{gcd}}]\!],[\![a,\ell_{\mathsf{gcd}}]\!],[\![(p+q-1),\ell_{\mathsf{gcd}}]\!])$. Note that this is fine because open $N=p\cdot q$ in MPC in the first steps of candidate generation to enforce input consistency.
4. Parties call $\mathcal{F}_{\mathsf{AdvMPC-CRT}}.\mathsf{Rand2k}$ with input $(\ell_{\mathsf{gcd}},3\lambda+4\sigma)$ to get a CRT sharing $[\![v,\ell_{\mathsf{gcd}}]\!]$ of a random value $v$, bounded by $2^{3\lambda+4\sigma}$.
5. All parties call $\mathcal{F}_{\mathsf{MPC\text{-}CRT}}.\mathsf{Add}([\![\hat{z},\ell_{\mathsf{gcd}}]\!],[\![z,\ell_{\mathsf{gcd}}]\!],[\![v\cdot N,\ell_{\mathsf{gcd}}]\!])$.
6. All parties call $\mathcal{F}_{\mathsf{MPC\text{-}CRT}}.\mathsf{Open}([\![\hat{z},\ell_{\mathsf{gcd}}]\!])$, and compute $\hat{z}=\mathsf{CRTrec}(\hat{z},\ell_{\mathsf{gcd}})$.
7. Locally check whether $\mathsf{gcd}(N,\hat{z})=1$. Otherwise parties output $\mathsf{Abort}$ and restart the protocol.

**Fig. 12.** RSA modulus generation protocol based on distributed sieving

**Theorem 3.** *The execution of the protocol $\Pi_{\mathsf{RSAGen}}$ UC-securely realizes the functionality $\mathcal{F}_{\mathsf{RSAGen}}$, in the $(\mathcal{F}_{\mathsf{ABBWithErrors}}, \mathcal{F}_{\mathsf{MPC}}, \mathcal{F}_{\mathsf{AgreeRandom}}, \mathcal{F}_{\mathsf{Broadcast}})$-hybrid model with statistical security against a static, active adversary that corrupts up to $n-1$ parties.*

*Proof Sketch.* Let $\mathcal{A}$ be a static malicious adversary, who interacts with the parties running $\Pi_{\mathsf{RSAGen}}$ and can corrupt up to $n-1$ parties. We construct a simulator $\mathcal{S}$, simulating the ideal functionality $\mathcal{F}_{\mathsf{RSAGen}}$, such that no environment $\mathcal{Z}$ can distinguish whether it is interacting with $\mathcal{A}$ and the $\Pi_{\mathsf{RSAGen}}$, or with $\mathcal{A}$ and $\mathcal{F}_{\mathsf{RSAGen}}$. 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 $\gcd(M_{\mathsf{sample}}, \hat{p}^{(j)}) = 1$.
2. For each honest $P_j \in \mathcal{H}$, $\mathcal{S}$ calls $\mathcal{F}_{\mathsf{ABBWithErrors}}.\mathsf{Input}(\hat{p}^{(j)})$. For each corrupt $P_j \in \mathcal{C}$, $\mathcal{S}$ receives $\mathcal{F}_{\mathsf{ABBWithErrors}}.\mathsf{Input}(\hat{p}^{(j)})$ from $\mathcal{A}$.
3. When all parties call $\mathcal{F}_{\mathsf{ABBWithErrors}}.\mathsf{Mult}$, $\mathcal{S}$ waits for $\Delta_p$ from $\mathcal{A}$. After receiving $\Delta_p$, $\mathcal{S}$ computes $p' = \Delta_p + \prod_{j=1}^{n} \hat{p}^{(j)}$.
4. When all parties call $\mathcal{F}_{\mathsf{ABBWithErrors}}.\mathsf{Share}(p')$, $\mathcal{S}$ 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 $p^{(j)} = p'^{(j)} + r^{(j)} \cdot M_{\mathsf{sample}}$ lies in the range $[2^{\lambda}, 2^{\lambda+\sigma}]$.
6. When each party calls $\mathcal{F}_{\mathsf{MPC\text{-}CRT}}.\mathsf{Input}(p^{(j)}, \ell_{\mathsf{c}})$ in Step 7, $\mathcal{S}$ receives from $\mathcal{A}$ the inputs $(p^{(j)}, \ell_{\mathsf{c}})$ for each corrupt $P_j \in \mathcal{C}$. With these, $\mathcal{S}$ can reconstruct $p^{(j)}$ for each corrupt $P_j$ and then compute $m_p = \sum_j p^{(j)} \mod M_{\mathsf{sample}}$ using also its simulated shares. It then sends $\mathsf{Sample}$ to $\mathcal{F}_{\mathsf{RSAGen}}$ 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 $\mathcal{F}_{\mathsf{MPC\text{-}CRT}}$ on behalf of the honest parties $P_j \in \mathcal{H}$.

By computing the residue of $p$ and $q$ modulo $M_{\mathsf{sample}}$ as influenced by $\mathcal{A}$ in Step 6 of the protocol, $\mathcal{S}$ ensures that the distribution of $N \bmod M_{\mathsf{sample}}$ produced by $\mathcal{F}_{\mathsf{RSAGen}}$ 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 $\mathcal{F}_{\mathsf{RSAGen}}$, as measured by Lemma 1, and identically distributed to the honest shares in a real execution.

**Combine:**

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

As the shares input by $\mathcal{A}$ are passed on $\mathcal{F}_{\mathsf{RSAGen}}$ for the generation of $N$, and as the shares simulated by $\mathcal{S}$ are statistically close to those sampled at random by $\mathcal{F}_{\mathsf{RSAGen}}$, the distribution of $N$ output by $\mathcal{F}_{\mathsf{RSAGen}}$ 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 $\mathcal{F}_{\mathsf{AgreeRandom}}$, $\mathcal{S}$ simulates the sampling of the public $\gamma$.
2. The simulator then queries $\mathcal{F}_{\mathsf{RSAGen}}.\mathsf{Jacobi}(\gamma)$ and receives $y$.
3. To simulate the broadcast calls, $\mathcal{S}$ samples $r_j$ at random to compute $y^{(j)} = \gamma^{r_j}$ 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 $\mathcal{F}_{\mathsf{ABBWithErrors}}$ 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 $\mathcal{F}_{\mathsf{Broadcast}}(y^{(j)})$, $\mathcal{S}$ computes the new value of $y' = \prod_j y^{(j)}$ and sends it to $\mathcal{F}_{\mathsf{RSAGen}}$.
5. If $\mathcal{S}$ receives $\mathsf{Abort}$ from $\mathcal{F}_{\mathsf{RSAGen}}$, it makes the simulated honest parties output $\mathsf{Abort}$.

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 $\mathcal{F}_{\mathsf{RSAGen}}$. 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 $\mathcal{F}_{\mathsf{RSAGen}}$.

**Consistency Check:**

1. When all parties call $\mathcal{F}_{\mathsf{AdvMPC-CRT}}.\mathsf{Rand2k}$, $\mathcal{S}$ samples a random $x < 2^{\mathrm{sec}}$ and receives the shares $(x^{(j)}, \ell_{\mathsf{Jac}})$ for each corrupt $P_j \in \mathcal{C}$ from $\mathcal{A}$. It then samples the remaining shares $(x^{(j)}, \ell_{\mathsf{Jac}})$ for honest $P_j \in \mathcal{H}$, such that $x = \mathsf{CRTrec}(x, \ell_1) = \sum_{j=1}^{n} (x^{(j)}, \ell_{\mathsf{Jac}})$.
2. When all parties call $\mathcal{F}_{\mathsf{MPC\text{-}CRT}}.\mathsf{Mult}([\![t, \ell_{\mathsf{Jac}}]\!], [\![x, \ell_{\mathsf{Jac}}]\!], [\![((N-p-q+1)/4), \ell_{\mathsf{Jac}}]\!])$, $\mathcal{S}$ internally executes the MPC multiplication.
3. When all parties call $\mathcal{F}_{\mathsf{AdvMPC-CRT}}.\mathsf{ConvInt}([\![t, \ell_{\mathsf{Jac}}]\!])$, $\mathcal{S}$ samples $t_{\mathsf{Int}}^{(j)}$ for each party $P_j$ such that $t = \sum_{j=1}^{n} t_{\mathsf{Int}}^{(j)}$, and sends them.
4. To simulate the broadcast calls, $\mathcal{S}$ modifies one of the honest shares $\gamma^{t_{\mathsf{Int}}^{(i)}}$ for $P_i \in \mathcal{H}$ such that $\gamma^{t_{\mathsf{Int}}^{(i)}} = y^x \cdot (\prod_{j \neq i} \gamma^{t_{\mathsf{Int}}^{(j)}})^{-1}$, where $y$ is the value given to $\mathcal{S}$ by $\mathcal{F}_{\mathsf{RSAGen}}$ during the Jacobi command and where $\mathcal{S}$ uses its internal values of $x$ and $p^{(j)}, q^{(j)}$ to compute $t_{\mathsf{Int}}^{(j)}$ of the corrupt parties. $\mathcal{S}$ then broadcasts $\gamma^{t_{\mathsf{Int}}^{(j)}}$ on behalf of the honest parties.
5. When all parties call $\mathcal{F}_{\mathsf{MPC\text{-}CRT}}.\mathsf{Open}([\![x, \ell_{\mathsf{Jac}}]\!])$, $\mathcal{S}$ simulates the opening and sends $\mathsf{Consistency\ Check}$ to $\mathcal{F}_{\mathsf{RSAGen}}$ on behalf of the corrupt parties.
6. To reply to $\mathcal{F}_{\mathsf{RSAGen}}$ about the abort, $\mathcal{S}$ checks whether $\prod_j \gamma^{t_{\mathsf{Int}}^{(j)}} = y^x$ using the values that were broadcast. If the equality fails, $\mathcal{S}$ sends $\mathsf{Abort}$ 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 $\gamma^{t_{\mathsf{Int}}^{(i)}}$ in Step 4 is statistically close, as measured by Lemma 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 $\gamma^{t_{\mathsf{Int}}^{(j)}}$, 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 $\mathcal{F}_{\mathsf{RSAGen}}$ 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 $\tilde{N} = (N-p-q+1)/4 \neq 0 \mod \phi(N)$ and that the adversary introduced an error $\Delta_j$ such that $y' = \gamma^{\tilde{N}+\Delta_j} = \pm 1 \mod N$. When the adversary cheats in the broadcast of the consistency check, the simulator computes $\gamma^{t_{\mathsf{Int}}+\Delta_t}$, where $\Delta_t \neq 0$ represents the error introduced by $\mathcal{A}$. Thus, for the equality to hold, $\mathcal{A}$ needs to commit to $\Delta_t$ during the broadcast, such that $\gamma^{x\tilde{N}+\Delta_t} = y^x = \gamma^{x\tilde{N}} \mod N$. Since the distribution of $x\tilde{N}$ is uniform with a min-entropy of $2^{-\mathrm{sec}}$, because of the sampling of $x$, $\mathcal{A}$ has probability at most $2^{-\mathrm{sec}+1}$ of finding the correct $\Delta_t$. We finally note that $\Delta_t$ cannot simply be the right value in the group of exponents with higher probability than this, because Rosser and Schoenfeld [RS62] showed that

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

where $\gamma$ is the Euler–Mascheroni constant. This gives us that $\log_2(\phi(N)) > \mathrm{sec}$, when $\log_2(N) > 2048$ and $\mathrm{sec} \sim 80$.

**GCD Test:**

1. When all parties call $\mathcal{F}_{\mathsf{AdvMPC-CRT}}.\mathsf{Rand2k}$ in Step 2, $\mathcal{S}$ samples a random $a < 2^{2\lambda+2\sigma}$ and receives the shares $(a^{(j)}, \ell_{\mathsf{gcd}})$ for each corrupt $P_j \in \mathcal{C}$ from $\mathcal{A}$. It then samples the remaining shares $(a^{(j)}, \ell_{\mathsf{gcd}})$ for honest $P_j \in \mathcal{H}$, such that $a = \mathsf{CRTrec}(a, \ell_{\mathsf{c}}) = \sum_{j=1}^{n} (a^{(j)}, \ell_{\mathsf{gcd}})$, and stores them.
2. When all parties call $\mathcal{F}_{\mathsf{MPC\text{-}CRT}}.\mathsf{Mult}([\![z, \ell_{\mathsf{gcd}}]\!], [\![a, \ell_{\mathsf{gcd}}]\!], [\![(p+q-1), \ell_{\mathsf{gcd}}]\!])$, $\mathcal{S}$ internally executes the MPC multiplication.

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

If $\gcd(N,p+q-1))=b$ then $b\,|\,z+v\cdot N$ and $b\,|\,z=a(p+q-1)$. Now, $\gcd(N,\hat{z})$ in the protocol can differ from $b=\gcd(N,p+q-1)$, if $\gcd(N,a)=b'>b$; thus, by sampling a random $\tilde{z}$ such that $\gcd(N,\tilde{z})=\max\{b,b'\}$, the simulator remains consistent with both $b$ and the probability that $\gcd(N,a)=b'>b$ 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 $\mathcal{F}_{\mathsf{RSAGen}}$. □

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

*Proof.* We can write $S=M_{\mathsf{sample}}(\lfloor S/M_{\mathsf{sample}}\rfloor+\epsilon)$ with $0\leq\epsilon<1$. When dividing the range of size $S$ into blocks of size $M_{\mathsf{sample}}$, the last block will not be complete (if $M_{\mathsf{sample}}$ does not divide $S$). When reducing the elements of $[2^{\lambda},2^{\lambda+\sigma})$ modulo $M_{\mathsf{sample}}$, 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 $x=aM_{\mathsf{sample}}+b$ with $a\in\mathbb{N}$ and $b\in[0,M_{\mathsf{sample}})$. As $p^{(j)}$ is computed as $p^{(j)}=p'^{(j)}+r^{(j)}\cdot M_{\mathsf{sample}}$ in Step 5 of the sampling phase, this implies:

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

as $p'^{(j)}$ is uniform in $[0,M_{\mathsf{sample}})$, because of $\mathcal{F}_{\mathsf{ABBWithErrors}}.\mathsf{Share}(p')$ 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

$$|X_1|=\epsilon M_{\mathsf{sample}}\cdot\left(\left\lfloor\frac{S}{M_{\mathsf{sample}}}\right\rfloor+1\right)\qquad\text{and}\qquad|X_2|=(1-\epsilon)M_{\mathsf{sample}}\cdot\left\lfloor\frac{S}{M_{\mathsf{sample}}}\right\rfloor.$$

This yields the following distance calculation:

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

□

## 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. [CCD$^+$20]. In the cases where a statistical security parameter $\sigma$ 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. [CCD$^+$20], 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 [Kel20] 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 [KOS16], whereas for the malicious case, which we need for building the MPC-CRT, we use LowGear [KPR18], with TopGear [BCS19] 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 [DKL$^+$13, BDST20], for which we get concrete costs by running the *hemi* protocol in MP-SPDZ [Kel20]. The results are given in Table 2 for the two party case, while in Table 3 we have results for the 16 party case, where we also compare them with the the protocol of Chen et al. [CCD$^+$20]. As shown in Table 2, for two parties, our protocol is a factor of 3.3-3.9 more communication-efficient than the state-of-the-art [CCD$^+$20] in the semi-honest case, and by a factor of 32-37 in the malicious case (ranging for different bit-lengths of the birprimes generated). For the 16 party case, the protocol of Chen et al. [CCD$^+$20] outperforms ours by a factor of approximately 2 in the semi-honest case. We left the corresponding cell in Table 3 empty, to avoid confusion, as the rest of the improvement factors refer to our work outperfoming that of Chen et al. Then again, for the malicious case and for 16 parties, our protocol improves the communication cost over the state-of-the-art [CCD$^+$20] 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. [CCD$^+$20], to achieve the same number of Sample iterations. Note that the product of the first 130 primes is 1019 bits long. Frankel et al. [FMY98] select $r^{(j)}$ in the sampling phase at random from $[0, 2^n / M_{\mathsf{sample}}]$ where $n$ was the desired bit-length of $p$.

For $\lambda$ 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, $M_{\ell_\mathsf{c}}$, the product of primes in which the biprimes live, needs to be of length at least $2\lambda + 2\sigma$ bits, which results in $\ell_\mathsf{c} = 18$ (i.e., we need 18 CRT components of 128 bits each). Similarly, we compute $\ell_\mathsf{Jac} = 21$ and $\ell_\mathsf{gcd} = 46$.

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 [KOS16]. Since the cost is quadratic in the field size, our ABBWithErrors will work over all the small primes composing $M_{\mathsf{sample}}$.
   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 4 amount to 0.264kbits. This makes step 3+4 having a cost of 17.556 kbits.
   The remaining cost here comes from the Input calls to $\mathcal{F}_{\mathsf{MPC\text{-}CRT}}$. 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 6 in the Sampling phase amount to 48.67 kbits. One iteration of this phase has a total cost of 66.23 kbits.
2. **Combine.** The cost per multiplication triple using $\mathcal{F}_{\mathsf{MPC\text{-}CRT}}$ amounts to 12.862 kbits per party. This brings the cost of one execution of Step 2 to 231 kbits. The opening (Step 4) takes another 2.176 bits. One iteration of this step has a total cost of 233 kbits.
3. **Jacobi test.** The cost of this step is simply $\log_2(N) \cdot n$, which is $2n \cdot \lambda$ or about 4 kbits.
4. **Consistency check.** This step begins with a call to LevelUp from $\ell_\mathsf{c}$ to $\ell_\mathsf{Jac}$, thus requiring $21 - 18 = 3$ inputs per party, as well as a multiplication and an opening in the additional CRT components. Summing up to a total of $8.1 + 38.6 + 0.4 = 47.1$ kbits. Next, parties call Rand2k, which costs sec bit generations in one of the CRT components, sec outputs in this CRT component, and sec inputs in all the CRT components per party, for a total cost of

| Scheme | [CCD$^+$20] | Ours | [CCD$^+$20] | Ours | [CCD$^+$20] | Ours | Improvement Factor Range |
|---|---|---|---|---|---|---|---|
| $\kappa$ | 1024 | 1024 | 1536 | 1536 | 2048 | 2048 | |
| semi-honest (MB) | 139 | 41.68 | 416 | 116.55 | 910 | 243.30 | 3.3-3.9× |
| malicious (GB) | 20.81 | 0.64 | 43.42 | 1.188 | 74.52 | 1.99 | 32.5-37.4× |

**Table 2.** Communication per party (two parties). For [CCD$^+$20] the cost of the semi-honest protocol is based on the use of the OT extension of Keller et al. [KOS16]. 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.

$\mathsf{sec}\cdot(12.862+20\cdot 2\cdot 1.35+0.128)=8575$ kbits for $\mathsf{sec}=128$. As before, the multiplication cost is simply $21\cdot 12.862=270.1$ kbits. The call to $\mathsf{ConvInt}$ requires a call to $\mathsf{Rand2k}$ and one opening, which amounts to $177084+2.7=177087$ kbits. Finally, the parties need to broadcast an element in $\mathbb{Z}_N$ and then open an element in all the $\ell_{\mathsf{Jac}}$ $\mathsf{CRT}$ components, which requires communicating 4.8 kbits. One iteration of this phase has a total cost of 185984 kbits.

5. **GCD test.** Here again we start with a call to $\mathsf{LevelUp}$, which costs 358.6 kbits, and then a call to $\mathsf{Rand2k}$ for a cost of 297387.2 kbits. Next is a multiplication on all the 46 $\mathsf{CRT}$ components for a total cost of 591.7 kbits. Second to last, we do a final call to $\mathsf{Rand2k}$ with bigger parameters, so the cost this time is 583999 kbits. Finally, we open $\hat{z}$ for 5.9 kbits. This phase thus requires a total of 882423 kbits.

   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 [RST$^+$19], each random bit $[\![b]\!]$ 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 $\mathsf{Rand2k}$ protocol, in order to generate $n_B$ shared bits with $n$ parties shared across $\ell$ engines will require $n_B\cdot n$ Input calls to each of the $\ell$ 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 $\mathcal{F}_{\mathsf{MPC}}.\mathsf{Input}(\sum_{k=0}^{15} 2^k b_k^{(j)} \bmod m_i)$, 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 [RST$^+$19] by simply scaling the random coefficients. To fit everything in a 128-bit prime $\mathsf{MPC}$ 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 [CGH$^+$18]. The cost analysis of such an instantiation can be seen in Table 6.

| Scheme | [CCD+20] | Ours | [CCD+20] | Ours | [CCD+20] | Ours | Improvement Factor Range |
|---|---|---|---|---|---|---|---|
| $\kappa$ | 1024 | 1024 | 1536 | 1536 | 2048 | 2048 | |
| semi-honest (GB) | 2.09 | 4.34 | 6.24 | 12.17 | 13.65 | 25.23 | |
| malicious (GB) | 1020 | 68.8 | 4734 | 153.20 | 8100 | 281.91 | 14.8-30.9× |

**Table 3.** Communication per party malicious case (16 parties). For [CCD+20] the cost of the semi-honest protocol is based on the use of the OT extension of Keller et al. [KOS16].

| $\kappa$ | 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 | 45.63 | 4296 | 67.63 | 8071 | 92.4 | 13029 |
| | | | expected cost to sample a biprime (GBytes) | | | |
| E[Iterations] | 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 $\sigma$ times.

| $\kappa$ | 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[Iterations] | 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.

| $\kappa$ | 1024 | 1536 | 2048 |
|---|---|---|---|
| megabytes | 105.26 | 222.99 | 401.452 |

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

## References

ACS02. Joy Algesheimer, Jan Camenisch, and Victor Shoup. Efficient computation modulo a shared secret with application to the generation of shared safe-prime products. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 417–432. Springer, Heidelberg, August 2002.

AFL$^{+}$16. Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 805–817. ACM Press, October 2016.

BBBF18. Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 757–788. Springer, Heidelberg, August 2018.

BCS19. Carsten Baum, Daniele Cozzo, and Nigel P. Smart. Using TopGear in overdrive: A more efficient ZKPoK for SPDZ. In Kenneth G. Paterson and Douglas Stebila, editors, *SAC 2019*, volume 11959 of *LNCS*, pages 274–302. Springer, Heidelberg, August 2019.

BDST20. Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. MOTION - A framework for mixed-protocol multi-party computation. Cryptology ePrint Archive, Report 2020/1137, 2020. `https://eprint.iacr.org/2020/1137`.

BF97. Dan Boneh and Matthew K. Franklin. Efficient generation of shared RSA keys (extended abstract). In Burton S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 425–439. Springer, Heidelberg, August 1997.

CCD$^{+}$20. Megan Chen, Ran Cohen, Jack Doerner, Yashvanth Kondi, Eysa Lee, Schuyler Rosefield, and abhi shelat. Multiparty generation of an RSA modulus. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 64–93. Springer, Heidelberg, August 2020.

CGH$^{+}$18. Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 34–64. Springer, Heidelberg, August 2018.

CHI$^{+}$20. Megan Chen, Carmit Hazay, Yuval Ishai, Yuriy Kashnikov, Daniele Micciancio, Tarik Riviere, Abhi Shelat, Muthuramakrishnan Venkitasubramaniam, and Ruihan Wang. Diogenes: Lightweight Scalable RSA Modulus Generation with a Dishonest Majority. *IACR Cryptol. ePrint Arch.*, 2020:374, 2020.

Des94. Yvo Desmedt. Threshold cryptography. *European Transactions on Telecommunications*, 5(4):449–457, July/August 1994.

Des98. Yvo Desmedt. Some recent research aspects of threshold cryptography (invited lecture). In Eiji Okamoto, George I. Davida, and Masahiro Mambo, editors, *ISW'97*, volume 1396 of *LNCS*, pages 158–173. Springer, Heidelberg, September 1998.

DF90. Yvo Desmedt and Yair Frankel. Threshold cryptosystems. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 307–315. Springer, Heidelberg, August 1990.

DKL$^{+}$13. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 1–18. Springer, Heidelberg, September 2013.

DM10. Ivan Damgård and Gert Læssøe Mikkelsen. Efficient, robust and constant-round distributed RSA key generation. In Daniele Micciancio, editor, *TCC 2010*, volume 5978 of *LNCS*, pages 183–200. Springer, Heidelberg, February 2010.

DOK$^{+}$20. Anders P. K. Dalskov, Claudio Orlandi, Marcel Keller, Kris Shrishak, and Haya Shulman. Securing DNSSEC keys via threshold ECDSA from generic MPC. In Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider, editors, *ESORICS 2020, Part II*, volume 12309 of *LNCS*, pages 654–673. Springer, Heidelberg, September 2020.

FLNW17. Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 225–255. Springer, Heidelberg, April / May 2017.

FLOP18. Tore Kasper Frederiksen, Yehuda Lindell, Valery Osheter, and Benny Pinkas. Fast distributed RSA key generation for semi-honest and malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 331–361. Springer, Heidelberg, August 2018.

FMY98. Yair Frankel, Philip D. MacKenzie, and Moti Yung. Robust efficient distributed RSA-key generation. In Brian A. Coan and Yehuda Afek, editors, *17th ACM PODC*, page 320. ACM, June / July 1998.

Gil99. Niv Gilboa. Two party RSA key generation. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 116–129. Springer, Heidelberg, August 1999.

GRR98. Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In Brian A. Coan and Yehuda Afek, editors, *17th ACM PODC*, pages 101–111. ACM, June / July 1998.

GRR$^+$16. Lorenzo Grassi, Christian Rechberger, Dragos Rotaru, Peter Scholl, and Nigel P. Smart. MPC-friendly symmetric key primitives. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 430–443. ACM Press, October 2016.

HKRW20. Lukas Helminger, Daniel Kales, Sebastian Ramacher, and Roman Walch. Multi-party revocation in sovrin: Performance through distributed trust. Cryptology ePrint Archive, Report 2020/724, 2020. `https://eprint.iacr.org/2020/724`.

HMR$^+$19. Carmit Hazay, Gert Læssøe Mikkelsen, Tal Rabin, Tomas Toft, and Angelo Agatino Nicolosi. Efficient RSA key generation and threshold paillier in the two-party setting. *Journal of Cryptology*, 32(2):265–323, April 2019.

HMRT12. Carmit Hazay, Gert Læssøe Mikkelsen, Tal Rabin, and Tomas Toft. Efficient RSA key generation and threshold Paillier in the two-party setting. In Orr Dunkelman, editor, *CT-RSA 2012*, volume 7178 of *LNCS*, pages 313–331. Springer, Heidelberg, February / March 2012.

Kel20. Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1575–1590, 2020.

KL20. Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. CRC press, 2020.

KOS16. Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 830–842. ACM Press, October 2016.

KPR18. Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 158–189. Springer, Heidelberg, April / May 2018.

MWB99. Michael Malkin, Thomas D. Wu, and Dan Boneh. Experimenting with shared generation of RSA keys. In *NDSS'99*. The Internet Society, February 1999.

Pie19. Krzysztof Pietrzak. Simple verifiable delay functions. In Avrim Blum, editor, *ITCS 2019*, volume 124, pages 60:1–60:15. LIPIcs, January 2019.

PS98. Guillaume Poupard and Jacques Stern. Generation of shared RSA keys by two parties. In Kazuo Ohta and Dingyi Pei, editors, *ASIACRYPT'98*, volume 1514 of *LNCS*, pages 11–24. Springer, Heidelberg, October 1998.

Rab98. Tal Rabin. A simplified approach to threshold and proactive RSA. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 89–104. Springer, Heidelberg, August 1998.

RS62. J. Barkley Rosser and Lowell Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois Journal of Mathematics*, 6(1):64 – 94, 1962.

RSA78. Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the Association for Computing Machinery*, 21(2):120–126, 1978.

RST$^+$19. Dragos Rotaru, Nigel P. Smart, Titouan Tanguy, Frederik Vercauteren, and Tim Wood. Actively secure setup for SPDZ. Cryptology ePrint Archive, Report 2019/1300, 2019. `https://eprint.iacr.org/2019/1300`.

Sha79. Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

ST06. Berry Schoenmakers and Pim Tuyls. Efficient binary conversion for Paillier encrypted values. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 522–537. Springer, Heidelberg, May / June 2006.

ST19. Nigel P. Smart and Younes Talibi Alaoui. Distributing any elliptic curve based protocol. In Martin Albrecht, editor, *17th IMA International Conference on Cryptography and Coding*, volume 11929 of *LNCS*, pages 342–366. Springer, Heidelberg, December 2019.

Wes19. Benjamin Wesolowski. Efficient verifiable delay functions. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part III*, volume 11478 of *LNCS*, pages 379–407. Springer, Heidelberg, May 2019.

# A Unauthenticated Arithmetic Black Box Protocols

$\Pi_{\mathsf{ABBWithErrors}}$

**Initialize:** The parties create the necessary preprocessing material for the online phase, receiving a number of multiplication triples $([a],[b],[c])$, and mask values $(r_i,[r_i])$.

**Input:** To share an input $x_i$, party $P_i$ takes an available mask value $(r_i,[r_i])$ and does the following:

1. Broadcast $\epsilon \leftarrow x_i - r_i$.
2. The parties compute $[x_i] \leftarrow [r_i] + \epsilon$.

**Multiply:** On input $([x],[y])$ the parties do the following:

1. Take one multiplication triple $([a],[b],[c])$, compute $[\epsilon] \leftarrow [x]-[a],[\rho] \leftarrow [y]-[b]$, and open these shares to get $\epsilon,\rho$, respectively.
2. Set $[z] \leftarrow [c] + \epsilon \cdot [b] + \rho \cdot [a] + \epsilon \cdot \rho$.

**Share:** On input $[x]$, each party $P_i$ retrieves its own share $x^{(i)}$ of $x$ from its local memory.

**Fig. 13.** Protocol for passively secure MPC adjusted for passive security from MASCOT [KOS16].

$\Pi_{\mathsf{InputTuple}}$

**Input:** On Input $(\mathsf{Input},P_j)$ from all parties do the following:

1. $P_j$ samples $r \xleftarrow{\$} \mathbb{F}$
2. All parties output $[r]$ and $P_j$ outputs $r$.

**Fig. 14.** Protocol for passively secure Input Tuples as presented in MASCOT [KOS16].

$\Pi_{\mathsf{TripleGeneration}}$

**Multiply:**

1. Each party samples $a^{(i)} \xleftarrow{\$} \mathbb{F}, b^{(i)} \xleftarrow{\$} \mathbb{F}$.
2. Every ordered pair of parties $P_i,P_j$ does the following:
   (a) Both parties call $\mathcal{F}_{\mathsf{ROT}}^{k,k}$, where $P_i$ inputs $(a_1^{(i)},...,a_k^{(i)}) = \mathbf{g}^{-1}(a^{(i)}) \in \mathbb{F}_2^k$.
   (b) $P_j$ receives $q_{0,h}^{(j,i)},q_{1,h}^{(j,i)} \in \mathbb{F}$, and $P_i$ receives $s_h^{(i,j)} = q_{a_h^{(i)},h}^{(j)}$, for $h=1,...,k$.
   (c) $P_j$ sends $d_h^{(j,i)} = q_{0,h}^{(j,i)} - q_{1,h}^{(j,i)} + b^{(j)}, h \in [k]$.
   (d) $P_i$ sets $t_h^{(i,j)} = s_h^{(i,j)} + a^{(i)} \cdot d_h^{(j,i)} = q_{0,h}^{(j,i)} + a_h^{(i)} \cdot b^{(j)}$, for $h=1,...k$. Set $q_h^{(j,i)} = q_{0,h}^{(j,i)}$.
   (e) $P_i$ sets $\mathbf{c}_{i,j}^{(i)} = \langle \mathbf{g},\mathbf{t} \rangle \in \mathbb{F}$, for $\mathbf{t}$ the above $k$-element vector.
   (f) $P_j$ sets $\mathbf{c}_{i,j}^{(j)} = -\langle \mathbf{g},\mathbf{q} \rangle \in \mathbb{F}$, for $\mathbf{q}$ the above $k$-element vector.
   (g) Now we have: $\mathbf{c}_{i,j}^{(i)} + \mathbf{c}_{i,j}^{(j)} = a^{(i)} \cdot b^{(j)} \in \mathbb{F}$
3. Each party $P_i$ computes: $\mathbf{c}^{(i)} = a^{(i)} \cdot b^{(j)} + \sum_{j \neq i} (\mathbf{c}_{i,j}^{(i)} + \mathbf{c}_{j,i}^{(i)})$

**Combine:**

1. Sample $r,\hat{r} \xleftarrow{\$} \mathcal{F}_{\mathsf{Rand}}(\mathbb{F})$.
2. Each party $P_i$ sets:
   (a) $a^{(i)} = \langle a^{(i)},r \rangle, c^{(i)} = \langle c^{(i)},r \rangle$, and
   (b) $\hat{a}^{(i)} = \langle a^{(i)},\hat{r} \rangle, \hat{c}^{(i)} = \langle c^{(i)},\hat{r} \rangle$

**Output:** $([a],[b],[c])$ as a valid triple.

**Fig. 15.** Protocol for Triple Generation adjusted for passive security from MASCOT [KOS16].

# B Standard functionalities

**Functionality $\mathcal{F}_{\mathsf{MPC}}$**

Let $[x]$ denote the identifier for a value $x$ stored in the functionality. Let $A \subset \{1,...,n\}$ denote the index set of the corrupted parties.

**Input:** Receive a value $x \in \mathbb{F}_p$ from some party and store $x$.

**Mult**$([x],[y])$**:** Compute $z = x \cdot y$ and store $[z]$.

**Share**$([x])$**:** For each $i \in A$ receive $x_i \in \mathbb{F}_p$ from the adversary. Sample uniform honest parties' shares ${x_j}_{j \notin A}$ s.t. $\sum_{i=1}^{n} x_i = x$. Send $x_i$ to $P_i$.

**Random:** Sample $r \overset{\$}{\leftarrow} \mathbb{F}_p$ and store $[r]$.

**Sum**$([x_1],...,[x_k])$**:** Compute $x = x_1 + \cdots + x_k$ and store $[x]$.

**GenBit():** Sample $b \overset{\$}{\leftarrow} \{0,1\}$ and store $[b]$.

**Open**$([x])$**:** Send the value $x$ to all parties.

**OpenTo**$([x],j)$**:** Send the value $x$ to party $P_j$.

**Fig. 16.** Arithmetic MPC Functionality

**Functionality $\mathcal{F}_{\mathsf{Broadcast}}$**

1. Receive a value $x$ from party $P_j$.
2. Send $x$ to all parties $P_i, i \neq j$.

**Fig. 17.** Broadcast Functionality

**Functionality $\mathcal{F}_{\mathsf{AgreeRandom}}$**

1. Receive a value $x_i$ from all parties $P_i$.
2. Compute $x = \sum x_i$.
3. Send $x$ to all parties $P_i$.

**Fig. 18.** Functionality to Agree on a Common Public Value

**Functionality $\mathcal{F}_{\mathsf{Rand}}$**

**Init**: On input $(\mathsf{Init}, \mathsf{sid}, \mathbb{F})$ from all parties await for incoming messages.

**Random**: On input $(\mathsf{Random}, \mathsf{sid})$ from all parties sample $r \overset{\$}{\leftarrow} \mathcal{U}(\mathbb{F})$ and send it to $\mathcal{S}$. Wait for $\mathcal{S}$ reply: if message is $\mathsf{OK}$ then send $r$ to all parties. If the message is $\mathsf{Abort}$ and send $\mathsf{Abort}$ to all parties and halt.

**Fig. 19.** Rand Functionality