Multiparty Generation of an RSA Modulus

Schuyler Rosefield
rosefield.s@northeastern.edu

February 28, 2020

Abstract

We present a new multiparty protocol for a distributed generation of an RSA modulus, with security against any subset of maliciously colluding parties.

At the center of our design is a new protocol template that generalizes in spirit many previous constructions, including Hazay et al. (JCRYPT'19) and Frederiksen et al. (Crypto'18). The new approach leads to a modular design with a simpler proof, as well as asymptotic and concrete efficiency improvements over the state-of-the-art protocols.

Our protocol makes use of a new sieving technique that overcomes an inherent leakage in the approach of Frederiksen et al. while also leading to simple and fast filtering of false candidates. In contrast to Hazay et al., our protocol is based only on oblivious transfer and the hardness of factoring, and does not rely on the ElGamal or Paillier homomorphic encryption schemes.

Contents

1 Introduction

An RSA modulus is a number $N = p \cdot q$ where p, q are primes. Shortly after the seminal RSA paper [RSA78], the problem of efficiently sampling such a modulus required custom ASICs [Riv80, Riv84]. By the mid '90s, however, the task was considered simple enough to perform on the average user's personal computer. The de facto method was to pick random κ-bit numbers (where κ is the security parameter), test primality using Miller-Rabin [Mil76, Rab80] until two primes were identified, and then multiply to generate a 2κ-bit modulus. This de facto method suffices when a single party wishes to create an RSA modulus such that no one else *knows* the corresponding factors (p, q).

Boneh and Franklin [BF97, BF01], in the context of *threshold cryptography*, initiated the study of *distributed* generation of an RSA modulus N by a set of parties such that no minority can factor N.[1] For multiparty RSA modulus generation, a clear observation is that applying generic MPC protocols to the de facto sampling method results in an infeasible solution. A Boolean circuit that mimics the de facto approach for sampling N has size $O(\kappa^4)$, since implementing the Miller-Rabin primality test requires repeatedly computing $a^{p-1} \pmod{p}$, where p is a *secret-shared* input to the secure computation.

Instead, Boneh and Franklin [BF97, BF01] constructed a biprimality test that generalizes Miller-Rabin. In particular, their biprimality test avoids computing modular exponentiations with secret moduli; instead, all exponentiations are carried out publicly modulo N, avoiding the need for generic MPC to securely evaluate a large Boolean circuit. With this insight, they introduced a three-phase structure to this secure sampling task that all subsequent works roughly embraced:

1. **Prime Candidate Sieving**: candidate values for p and q are sampled jointly in secret-shared form, and a weak-but-cheap form of trial division sieves the candidates, identifying *likely* primes.

2. **Modulus Reconstruction**: the modulus $N = p \cdot q$ is securely computed and revealed.

3. **Biprimality Testing**: using a distributed protocol, test whether N is a biprime (that is, a composite of two primes) with overwhelming probability. If N is not a biprime, then restart at prime candidate sieving.

Boneh and Franklin worked in the semi-honest n-party model with an honest majority, using the BGW protocol [BGW88] for most tasks. Their work lead to many follow-up works (see Section 1.3). Most notably, recent two works managed to extend this approach to protocols in malicious setting.

[1] Prior works generally consider RSA *key generation* and include steps for generating shares of (e, d) such that $e \cdot d \equiv 1 \pmod{\varphi(N)}$. This work focuses on the task of sampling the RSA modulus N; since prior techniques can be applied to sample (e, d) after sampling N, and a distributed generation of an RSA modulus on its own has many applications such as Verifiable Delay Functions, we omit the discussion of this from the paper.

Hazay et al. [HMRT12, HMR$^+$19] constructed a maliciously secure two-party protocol (that extends to the multiparty setting). Their key insight is that both the sieving by small primes and the multiplication of p and q can be computed securely using additive-homomorphic encryption. They rely both on the ElGamal and Paillier encryption schemes and achieve malicious security through a variety of zero-knowledge proofs for ElGamal- and Paillier-based relations. Thus, while their protocol represents substantial progress in implementation, their approach requires *extra* complexity assumptions for additive homomorphisms, intricate analysis, use of many custom zero-knowledge proofs, and a large complexity analysis.

Frederiksen et al. [FLOP18] presented a maliciously secure two-party protocol, relying mainly on oblivious transfer (OT). They claim to achieve malicious security at only constant overhead on top of the semi-honest variant of their protocol. Their main insight is to use the OT-based multiplier proposed by Gilboa [Gil99] and to also perform sieving using OT. Their approach for achieving malicious security follows the folklore technique in which a proof of honesty is added as the last step. Their protocol is highly tailored to the two-party case; it is not clear how to extend their OT-based sieving or proof of honesty in a straightforward way to the multiparty case.

The approach of [FLOP18] leads to a efficient two-party protocols that can be implemented; however, it suffers from two shortcomings. First, due to selective-failure attacks, their sieving method *leaks* certain information to the adversary. In addition to providing weaker security guarantees, integrating this leakage in the ideal functionality leads to complicated analysis. Second, their protocol permits a malicious adversary to *covertly* induce false negatives, i.e., sabotage candidates that actually are biprimes without being detected. This affects both *security*, as the adversary can rejection sample biprimes based on the additional leaked information, and on *efficiency*, as this adds a caveat to their claim of constant overhead for the malicious protocol compared to their semi-honest variant. In particular, in order for an honest party to be convinced that the sampling failures are adversarial, sufficiently many instances must be executed (sequentially or in parallel) to reduce probability of statistical sampling failures.

Despite the security and efficiency advancements of [HMR$^+$19, FLOP18], it still remains unclear how to efficiently sample an RSA modulus in a distributed way without relying on “heavy” cryptographic primitives such as Paillier encryption. In the two-party case it is unclear how to efficiently sample an RSA modulus without leaking additional information on the primes and without allowing covert rejection sampling. In this work, we show how to efficiently achieve both of these tasks.

1.1 Our Results

Our main contribution is a new protocol template for a multiparty sampling of an RSA modulus. The protocol template follows the approach of [BF01] and generalizes in spirit both [HMR$^+$19] and [FLOP18]. The new approach leads to a modular design with a simpler proof and eliminates the covert attacks and inherent leakage from [FLOP18]. Further, we show how to efficiently instantiate the "building blocks" of the protocol template, assuming merely OT. This leads to asymptotic and concrete efficiency improvements over the state-of-the-art.

The protocol template is defined in the $(\mathcal{F}_{\mathsf{AugMul}}, \mathcal{F}_{\mathsf{Biprime}})$-hybrid model, where the *augmented multiplication* functionality $\mathcal{F}_{\mathsf{AugMul}}$ and the *biprimality-test* functionality $\mathcal{F}_{\mathsf{Biprime}}$ are ideally computed. The main technical contribution is that we show how to use the CRT for improved security and efficiency over prior works, as well as a framework for modular construction and analysis.

Loosely speaking, $\mathcal{F}_{\mathsf{AugMul}}$ allows the parties to samples additive shares of nonzero values modulo small primes, and to multiply such shares to get shares of the product. This allows the parties to sample shares of p and q that are not divisible by small primes. $\mathcal{F}_{\mathsf{Biprime}}$ receives the shares of p and of q (over the integers) from the parties and checks whether p and q are both primes. We denote by $\mathcal{F}_{\mathsf{RSAGen}}$ the modulus-sampling functionality that samples p and q from the same distribution of [BF01] and return either $N = p \cdot q$ in case p and q are primes, or a sampling-failure indicator otherwise.

Theorem 1.1 (protocol template, malicious security, informal)**.** *Assuming the hardness of factoring, $\mathcal{F}_{\mathsf{RSAGen}}$ can be securely computed with abort in the $(\mathcal{F}_{\mathsf{AugMul}}, \mathcal{F}_{\mathsf{Biprime}})$-hybrid model, tolerating a PPT malicious adversary that may corrupt any subset of the parties.*

Surprisingly, in the semi-honest setting, the protocol template *perfectly* realizes $\mathcal{F}_{\mathsf{RSAGen}}$. This holds since we only need to rely on the hardness of factoring in case the adversary cheats.

Theorem 1.2 (protocol template, semi-honest security, informal)**.** *$\mathcal{F}_{\mathsf{RSAGen}}$ can be securely computed with guaranteed output delivery and with perfect correctness in the $(\mathcal{F}_{\mathsf{AugMul}}, \mathcal{F}_{\mathsf{Biprime}})$-hybrid model, tolerating a computationally unbounded semi-honest adversary that may corrupt any subset of the parties.*

1.2 Our Techniques

Our protocol is the best of the previous two recent works: we present a multiparty maliciously-secure protocol (against $n-1$ parties) that only relies on OT and is conceptually simpler, achieves stronger security that doesn't leak extra information, and is asymptotically and concretely more efficient than all prior works. We proceed to describe our technical contributions in greater details.

Constructive sampling. All prior works sample *destructively*. That is, they first sample $p, q \in [0, 2^{\kappa}]$, do limited distributed trial division of p, q by small primes, and multiply to produce a 2κ-bit N. If the ℓ^{th} trial division fails, then p, q are discarded (as is all of the work up to this point and the first $\ell - 1$ trial divisions).

Contrastingly, our technique leverages the Chinese Remainder Theorem (CRT) to *constructively* sample N. We sample p and q in CRT-form with respect to the first $\kappa / \ln \kappa$ odd primes. For example, to ensure that p is not divisible by the 3, players sample an element of $\mathbb{Z}_3^*$, and jointly check that the sum of their elements is nonzero.

We also use the CRT algorithm to save asymptotically on the cost of multiplying large numbers. Specifically, multiplying two κ-bit values using OT-based protocols (e.g., Gilboa [Gil99]) conventionally requires κ OTs with messages of size κ each. This approach fundamentally requires $O(\kappa^2)$ work and communication, even in the semi-honest case, and $O(\kappa^2 + \kappa s)$ in the malicious case [DKLS19] (for s bits of statistical security) with state of the art OT extension protocols [BCG$^+$19].

Instead, consider the CRT-representation of two κ-bit numbers with respect to the set of primes $\{m_1, \ldots, m_\ell\}$, where each m_i is roughly $L = \log \kappa$ bits (and so $\ell = \kappa / \log \kappa$). Multiplying the CRT-representation requires ℓ secure multiplications of L-bit values instead; plugging in the relationships between κ, s, ℓ, and L, we obtain a cost of $O(\kappa \log \kappa)$ in the semi-honest setting and $O(\kappa \log \kappa + \kappa s)$ for the malicious variant. This provides an asymptotic improvement over [Gil99] (for semi-honest) and [FLOP18] (for malicious) by factors of $\kappa / \log \kappa$.

Standard security with abort. Our CRT representation also provides a key step in our malicious security analysis. Our protocol realizes the *standard* notion of malicious security with abort, i.e., the adversary can instruct the functionality to abort even when a biprime is successfully sampled, but honest parties are always made aware of this fact. We accomplish this by revealing all inputs when the sampling fails (also suggested in [Gav12, HMR$^+$19]) and verifying in the clear that the candidate was indeed not a biprime/false negative.

Proving security for this simple idea turns out to be surprisingly subtle. The challenge is that a simulator needs to simulate the protocol transcript for the multiplications of the honest parties without knowing their inputs. Later, if all inputs need to be revealed, the simulator has to "explain" the simulated transcript for honest-parties multiplications (to a corrupt counterparty).

A naïve implementation of this idea therefore requires the protocol to be secure against adaptive corruptions. Another standard-but-inefficient technique is instead of revealing secret values to prove honest behaviour in using zero-knowledge proofs. To our knowledge, there is no efficient instantiation of such an equivocal multiplier based on OT. The technical bottleneck is that in order to defend against selective failure attacks, recent maliciously secure OT multipliers [DKLS18, FLOP18] instruct the receiver of the OT (Bob) to use a high-entropy encoding of its input. However, given Bob's input for which a simulated transcript has to be "explained," it is unclear how to work backwards and obtain the randomness used to produce such an encoding. Intuitively, this task closely resembles that of solving a random instance of subset sum.

The reason why our approach overcomes this difficulty is that the multiplications are performed component-wise, where each component is of size $O(\log \kappa)$ bits; the simulator can therefore simply guess random encodings until finding one that maps to the required element. We show that this strategy: (1) succeeds in strict polynomial time as the target field is small, and (2) induces a distribution statistically close to the real execution (using a lemma from [IN96]).

Overall, we abstract this multiplication technique into the ideal functionality $\mathcal{F}_{\mathsf{AugMul}}$ and show that this form of “privacy-free” malicious security (where honest behaviour is verified at the cost of sacrificing privacy) leads to considerable efficiency gains: for small fields it is up to a multiplicative factor of s cheaper than checking with privacy as usually done in OT-based multipliers [KOS16, DKLS18]. As the bulk of the candidates generated are discarded, verifying that they were processed honestly with this privacy-free check results in substantial savings.

Conceptual simplification. The protocol template is defined via modular approach that abstracts away implementation details of $\mathcal{F}_{\mathsf{AugMul}}$ and $\mathcal{F}_{\mathsf{Biprime}}$. We consider this framework a major contribution to this active research area.

In particular, the only differences between our semi-honest and malicious protocols are: (1) perform the consistency check at the protocol-template level (either a privacy-preserving check if the result is a biprime or a privacy-free check if it is a non-biprime), and (2) instantiate the two functionalities $\mathcal{F}_{\mathsf{AugMul}}$ and $\mathcal{F}_{\mathsf{Biprime}}$ with maliciously secure primitives. In contrast, the monolithic analysis of malicious security in both [HMR$^+$19, FLOP18] is complicated and harder to follow. Both protocols require many additional steps compared to the semi-honest protocol, including a separate proof of honesty in the latter case.

The modular approach may also lead to protocols that are secure in stronger security models. One example, presented in Theorem 1.2, is perfect security in the semi-honest setting. Another simple observation is that instantiating $\mathcal{F}_{\mathsf{AugMul}}$ and $\mathcal{F}_{\mathsf{Biprime}}$ with adaptively secure primitives will lead to an RSA modulus-sampling protocol that is adaptively secure. This is in contrast to [HMR$^+$19, FLOP18], where the security is inherently static. Similarly, the protocol template can be easily extended to provide security with identifiable abort, where cheating parties are identified by the honest parties. We do not provide the proofs for these extensions in this paper, in order to focus on the conceptual contributions of the framework.

Another advantage of our modular analysis is that future works can simply swap out the implementations of $\mathcal{F}_{\mathsf{AugMul}}$ or $\mathcal{F}_{\mathsf{Biprime}}$. For example, a follow-up work by Chen et al. [CIK$^+$20] replaces our OT-based multiplier with a multiplier built with an RLWE-based additive-homomorphic encryption.

Maliciously secure biprimality test. To instantiate a maliciously secure version of the Boneh-Franklin biprimality test, we use a generic maliciously secure MPC protocol to recompute the modulus and perform range checks on the inputs. A key insight in our protocol is our reduction to show that if a party is able to cheat in this step by supplying *incorrect inputs* to $\mathcal{F}_{\mathsf{Biprime}}$ (relative to the candidate biprime N that was generated in the outer RSA protocol) for which $\mathcal{F}_{\mathsf{Biprime}}$ returns a positive answer (that the candidate is a biprime), then they can essentially factor N. We are careful to rely on this only for *successful* biprime candidates (and ensure that sampling success/failure is not in the hands of the adversary) and substantiate this claim by a reduction to factoring biprimes produced by $\mathcal{F}_{\mathsf{RSAGen}}$.

1.3 Additional Related Work

Frankel, MacKenzie, and Yung [FMY98] adjusted [BF97] to remain secure against malicious adversaries, albeit still in the honest-majority setting. Their main contribution is explaining how to perform robust distributed multiplication over the integers. Cocks [Coc97] proposed a method for two-party (and multi-party) RSA key generation under heuristic assumptions, and latter attacks by Coppersmith (see [Coc98]) and Joye and Pinch [JP99] suggest privacy may be compromised in certain situations. Poupard and Stern [PS98] presented a maliciously secure two-party protocol based on oblivious transfer (OT). Gilboa [Gil99] focused on improving the efficiency of the semi-honest two-party model and introduced a novel method to multiply $p \cdot q$ using oblivious transfer.

Algesheimer, Camenish, and Shoup [ACS02] described a method to compute modular exponentiations modulo p using secret-sharing conversion techniques that rely on computing approximations of $1/p$. As a result, they could independently sample primes p and q. However, each invocation of their Miller-Rabin test still requires $O(\kappa^3)$ bit operations per party, and their overall protocol requires $O(\kappa^5/\log^2 \kappa)$ bit operations per party and $\Theta(\kappa)$ rounds of interaction. In particular, to sample a $\kappa = 2000$-bit modulus, the estimates are 10,000 rounds of communication making this method unwieldy (see [DM10]). Damgård and Mikkelsen [DM10] extended their work to improve both complexity and rounds by several orders, while also designing a maliciously-secure variant in the honest-majority setting based on replicated secret sharing. They reconstructed N before the primality tests, with the key insight that the public knowledge of N can be used to simplify the distributed primality test. Their protocol is at least a factor of $O(\kappa)$ better than [ACS02] but still requires *hundreds* of rounds; we were not able to compute an explicit complexity analysis of their approach.

Organization. Basic notation and background information are given in Section 2. The ideal RSA modulus generation functionality is defined in Section 3, and the protocol template realizing it is given in Section 4. In Section 5, we present a biprimality-testing protocol, and in Section 6 the efficiency analysis of our protocol. Due to space limits, some of the details and proofs are deferred to the appendices.

2 Preliminaries

We use $=$ for equality, $:=$ for assignment, $\leftarrow$ for sampling from a distribution, $\equiv$ for congruence, $\approx_{\mathrm{c}}$ for computational indistinguishability, and $\approx_{\mathrm{s}}$ for statistical indistinguishability. In general, single-letter variables are set in *italic* font, multi-letter variables and function names are set in sans-serif font, and string literals are set in `slab-serif` font. mod is used to indicate the modulus operator, while $\pmod{m}$ at the end of a line is used to indicate that all equivalence relations on that line are to be taken over the integers modulo m. By convention, we parameterize computational security by the bit-length of each prime in an RSA biprime; we denote this length by κ throughout. Likewise, we use s to represent the statistical parameter.

Vectors and arrays are given in bold and indexed by subscripts; thus $\mathbf{x}_i$ is the i^{th} element of the vector $\mathbf{x}$, which is distinct from the scalar variable x. When we wish to select a row or column from a two-dimensional array (or a plane from a three-dimensional array, etc), we place a $*$ in any dimension along which we are not selecting. Thus $\mathbf{y}_{*,j}$ is the j^{th} column of matrix $\mathbf{y}$, and $\mathbf{y}_{j,*}$ is the j^{th} row. We use $\mathcal{P}_i$ to denote the party with index i, and when only two parties are present, we refer to them as Alice and Bob. Variables may often be subscripted with an index to indicate that they belong to a particular party. When arrays are owned by a party, the party index always comes first. We use $|x|$ to denote the bit-length of x, and $|\mathbf{y}|$ to denote the number of elements in the vector $\mathbf{y}$.

We prove our protocols secure in the Universal Composablility (UC) framework, and use standard UC notation. In Appendix B, we give a high-level overview and refer the reader to [Can01] for a further details. In functionality descriptions, we leave some standard bookkeeping elements implicit. For example, we assume that the functionality aborts if a party tries to reuse a session identifier, send messages out of order, etc. For convenience, we provide a function GenSID, which takes *any* number of arguments and deterministically derives a unique output (to be used as a Session ID) from those arguments.

For completeness, we describe the CRTRecon algorithm that given $\mathbf{m} = (\mathbf{m}_1, \ldots, \mathbf{m}_\ell)$ and $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_\ell)$ finds the unique $y \in \mathbb{Z}_M$.

Algorithm 2.1. $\mathsf{CRTRecon}(\mathbf{m} \in \mathbb{Z}^\ell, \mathbf{x} \in \mathbb{Z}_{\mathbf{m}_1} \times \cdots \times \mathbb{Z}_{\mathbf{m}_\ell})$

1. Compute $M = \prod_{j \in [\ell]} \mathbf{m}_j$.
2. For $j \in [\ell]$, compute $\mathbf{a}_j := M/\mathbf{m}_j$ and find $\mathbf{b}_j$ satisfying $\mathbf{a}_j \cdot \mathbf{b}_j \equiv 1 \pmod{\mathbf{m}_j}$ using the Extended Euclidean Algorithm (see [Knu69]).
3. Output $y := \sum_{j \in [\ell]} \mathbf{a}_j \cdot \mathbf{b}_j \cdot \mathbf{x}_j \bmod M$.

3 Assumptions and Ideal Functionalities

In this section, we define the ideal RSA modulus-sampling functionality. In Section 3.1, we define the precise factoring assumption that is considered, and in Section 3.2, the RSA modulus-sampling algorithm and the ideal functionality.

3.1 Factoring Assumptions

The standard factoring experiment (Experiment 3.1) as formalized by Katz and Lindell [KL15] is parameterized by an adversary $\mathcal{A}$ and a biprime-sampling algorithm $\mathsf{GenModulus}$. On input 1^κ, this algorithm returns (N, p, q), where $N = p \cdot q$, and p and q are two κ-bit primes, except with probability negligible in κ.[2]

Experiment 3.1. $\mathsf{Factor}_{\mathcal{A},\mathsf{GenModulus}}(\kappa)$

1. Run $(N, p, q) \leftarrow \mathsf{GenModulus}(1^\kappa)$.
2. Send N to $\mathcal{A}$, and receive $p', q' > 1$ in return.
3. Output 1 if and only if $p' \cdot q' = N$.

In many cryptographic applications, $\mathsf{GenModulus}(1^\kappa)$ is defined to sample p and q uniformly from the set of primes in the range $[2^{\kappa-1}, 2^\kappa)$ [Gol01], and the factoring assumption with respect to this common $\mathsf{GenModulus}$ function states that for all PPT adversaries $\mathcal{A}$ there exists a negligible function negl such that

$$\Pr\left[\mathsf{Factor}_{\mathcal{A},\mathsf{GenModulus}}(\kappa) = 1\right] \leq \mathsf{negl}(\kappa).$$

Because *efficiently* sampling according to this uniform distribution is difficult in a multiparty context, most prior works sample according to a different distribution, and thus using the moduli they produce requires a slightly different factoring assumption than the traditional one. In particular, several recent works use a distribution originally proposed by Boneh and Franklin [BF01], which is especially well-adapted to multiparty sampling. Our work follows this pattern.

Boneh and Franklin's distribution is defined by the sampling algorithm BFGM, which takes as an additional parameter the number of parties n. The algorithm samples n integer shares, each in the range $[0, 2^{\kappa-\log n})$,and sums these shares to arrive at a candidate prime. This does *not* induce a uniform distribution on the set of κ-bit primes. Furthermore, BFGM only samples individual primes p or q that have $p \equiv q \equiv 3 \pmod 4$, in order to facilitate efficient distributed primality testing, and it filters out the subset of otherwise-valid moduli $N = p \cdot q$ that have $p \equiv 1 \pmod q$ or $q \equiv 1 \pmod p$.[3] We recall the formal description for completeness in Appendix A.

Any protocol whose security depends upon the hardness of factoring moduli output by our protocol (including our protocol itself) must rely upon the assumption that for every PPT adversary $\mathcal{A}$,

$$\Pr\left[\mathsf{Factor}_{\mathcal{A},\mathsf{BFGM}}(n, \kappa) = 1\right] \leq \mathsf{negl}(\kappa)$$

3.2 Our Distributed Modulus-Sampling Functionality

Unfortunately, our ideal modulus-sampling functionality cannot merely call BFGM; we wish our functionality to run in *strict* polynomial time, whereas

[2] We must allow a sampling failure in a negligible fraction of cases in order to ensure the algorithm completes in strict polynomial time.

[3] Boneh and Franklin actually propose two variations, one of which has no false negatives; we choose the other variation, as it leads to a more efficient sampling protocol.

the running time of BFGM is only *expected* polynomial. Thus, we define a new sampling algorithm, CRTSample, which might fail, but conditioned on success outputs samples statistically close to BFGM.[4] Furthermore, we give CRTSample a specific distribution of failures, which is tied to the design of our protocol. As a second concession to our protocol design (and following [HMR+19]), CRTSample takes as input up to $n-1$ integer shares of p and q, arbitrarily determined by the adversary, while the remaining shares are sampled randomly. We start by defining useful notation.

Definition 3.2 ((κ, n)-near-primorial vector)**.** *Let m_i be the i'th prime number and let ℓ be the largest integer such that $\prod_{i\in[\ell]} m_i < 2^{\kappa-\log n-1}$. Then, $\mathbf{m} = (4, m_2, \cdots, m_\ell)$ is the* (κ, n)-near-primorial vector*.*

Definition 3.3 ($\mathbf{m}$-coprimality)**.** *Let $\mathbf{m}$ be a vector of integers. An integer x is* $\mathbf{m}$-coprime *if and only if it is not divisible by any $\mathbf{m}_i$ for $i \in [|\mathbf{m}|]$.*

Algorithm 3.4. CRTSample$(n, \kappa, \{(p_i, q_i)\}_{i\in\mathbf{P}^*})$

1. Let $\mathbf{m} = (\mathbf{m}_1, \ldots, \mathbf{m}_\ell)$ be a (κ, n)-near-primorial vector and let $M := \prod_{i\in[\ell]} \mathbf{m}_i$.
2. For $i \in [n] \setminus \mathbf{P}^*$, sample $p_i \leftarrow [0, M)$ and $q_i \leftarrow [0, M)$ subject to
$$p_i \equiv q_i \equiv \begin{cases} 3 \pmod 4 & \text{if } i = 1 \\ 0 \pmod 4 & \text{if } i \neq 1 \end{cases}$$
and subject to p and q being $\mathbf{m}$-coprime, where $p := \sum_{i\in[n]} p_i$ and $q := \sum_{i\in[n]} q_i$ are computed over the integers.
3. If $\gcd(p\cdot q, p+q-1) = 1$, and if both p and q are primes, and if $p \equiv q \equiv 3 \pmod 4$, then output $(\texttt{success}, p, q)$; otherwise, output $(\texttt{failure}, p, q)$.

Boneh and Franklin [BF01, Lemma 2.1] showed that even given $n-1$ integer shares of the factors p, q does not provide any meaningful advantage in factoring biprimes from the distribution produced by BFGM and, by extension, CRTSample. Hazay et al. [HMR+19, Lemma 4.1] extended this argument to the malicious setting where the adversary is allowed to choose its own shares.

Lemma 3.5 ([BF01, HMR+19])**.** *Let $n < \kappa$. Suppose that there exists a pair of PPT algorithms $\mathcal{A}_1, \mathcal{A}_2$ as follows: For $(\mathsf{state}, \{(p_i, q_i)\}_{i\in[n-1]}) \leftarrow \mathcal{A}_1(n, \kappa)$, let N be a biprime sampled by running CRTSample$(n, \kappa, \{(p_i, q_i)\}_{i\in[n-1]})$, then $\mathcal{A}_2(\mathsf{state}, N)$ outputs the factors of N with probability at least $1/\kappa^d$. Then, there exists an expected-polynomial-time algorithm $\mathcal{B}$ that succeeds with probability $1/2^4n^3\kappa^d$ in the experiment $\mathsf{Factor}_{\mathcal{B},\mathsf{BFGM}}(\kappa, n)$.*

[4] CRTSample never outputs biprimes with factors smaller than κ, whereas BFGM outputs such biprimes with negligible probability.

Multiparty Functionality. Our ideal functionality $\mathcal{F}_{\mathsf{RSAGen}}$ is a natural embedding of $\mathsf{CRTSample}$ in a multiparty functionality: it receives inputs $\{(p_i, q_i)\}_{i\in\mathbf{P}^*}$ from the adversary and runs a single iteration of $\mathsf{CRTSample}$ with these inputs when invoked, and either outputs the corresponding modulus $N := p \cdot q$ if it is valid, or indicates that a sampling failure has occurred.

Running a single iteration of $\mathsf{CRTSample}$ per invocation of $\mathcal{F}_{\mathsf{RSAGen}}$ enables significant freedom in both use of $\mathcal{F}_{\mathsf{RSAGen}}$ and analysis of the protocol π_{RSAGen} that realizes it. In particular the protocol analysis is made independent of the success rate of the sampling procedure, and $\mathcal{F}_{\mathsf{RSAGen}}$ can be composed in different ways to tune the trade-off between resource usage and execution time.

The functionality may not deliver N to honest parties for one of two reasons: either $\mathsf{CRTSample}$ failed to sample a biprime, or the adversary caused the computation to abort. In both cases, the honest parties are informed of the cause of the failure, and consequently the adversary is unable to conflate the two cases. This is essentially the standard notion of security with abort, applied to the multiparty computation of the $\mathsf{CRTSample}$ algorithm. In both cases, the p and q output by $\mathsf{CRTSample}$ are given to the adversary. This leakage simplifies our proof considerably, and we consider it benign, since the honest parties never receive (and therefore cannot possibly use) N.

Functionality 3.6. $\mathcal{F}_{\mathsf{RSAGen}}(\kappa, n)$. **Distributed Biprime Sampling**

This n-party functionality attempts to sample an RSA modulus with prime length κ, and interacts directly with an ideal adversary $\mathcal{S}$ which corrupts parties indexed by $\mathbf{P}^*$. Let M be the largest number such that $M/2$ is a primorial number and $M < 2^{\kappa-\log_2(n)}$.

Sampling: On receiving $(\texttt{sample}, \mathsf{sid})$ from each party $\mathcal{P}_i$ for $i \in [n] \setminus \mathbf{P}^*$ and $(\texttt{adv-sample}, \mathsf{sid}, i, p_i, q_i)$ from $\mathcal{S}$ for $i \in \mathbf{P}^*$, if $0 \le p_i < M$ and $0 \le q_i < M$ for all $i \in \mathbf{P}^*$, then run $\mathsf{CRTSample}(n, \kappa, \{(p_i, q_i)\}_{i\in\mathbf{P}^*})$, and receive as a result either $(\texttt{success}, p, q)$ or $(\texttt{failure}, p, q)$.

- If $p \not\equiv 3 \pmod 4$ or $q \not\equiv 3 \pmod 4$, then send $(\texttt{factors}, \mathsf{sid}, p, q)$ to $\mathcal{S}$ and abort, informing all parties in an adversarially delayed fashion.
- If $p \equiv q \equiv 3 \pmod 4$, and the result was $\texttt{failure}$, then store $(\texttt{non-biprime}, \mathsf{sid}, p, q)$ in memory and send $(\texttt{factors}, \mathsf{sid}, p, q)$ to $\mathcal{S}$.
- If $p \equiv q \equiv 3 \pmod 4$, and the result was $\texttt{success}$, then compute $N := p \cdot q$, store $(\texttt{biprime}, \mathsf{sid}, N, p, q)$ in memory, and send $(\texttt{biprime}, \mathsf{sid}, N)$ to $\mathcal{S}$.

Output: On receiving either $(\texttt{proceed}, \mathsf{sid})$ or $(\texttt{cheat}, \mathsf{sid})$ from $\mathcal{S}$, if $(\texttt{biprime}, \mathsf{sid}, N, p, q)$ or $(\texttt{non-biprime}, \mathsf{sid}, p, q)$ exists in memory,

- If $\texttt{proceed}$ was received, then send either $(\texttt{biprime}, \mathsf{sid}, N)$ or $(\texttt{non-biprime}, \mathsf{sid})$ to all parties as adversarially delayed private output, as appropriate. Terminate successfully.

- If `cheat` was received, then abort, notifying all parties in an adversarially delayed fashion, and send $(\texttt{factors}, \mathsf{sid}, p, q)$ directly to $\mathcal{S}$.

Regardless, ignore all further instructions with this sid.

4 The Distributed Modulus-Sampling Protocol

In this section, we present the distributed RSA modulus-sampling protocol template. In Section 4.1, we define the ideal functionalities $\mathcal{F}_{\mathsf{AugMul}}$ and $\mathcal{F}_{\mathsf{Biprime}}$ that are used in the protocol template, and in Section 4.2, we define the protocol itself. Next, in Section 4.3, we present the security proof. We begin with a high-level overview of the construction.

High-level overview. As described in the Introduction, our distributed RSA protocol follows the template introduced by Boneh and Franklin [BF01]. The main differences compared to other protocols that follow this template (e.g., [HMR$^+$19, FLOP18]) is the novel use of CRT-based sampling that provides better efficiency and eliminates the inherent leakage in [FLOP18], and the modular approach that enables a relatively simple description of the maliciously secure protocol – almost as simple as the semi-honest version.

The protocol template consists of three phases:

Candidate sieving: The protocol is parametrized with a set "small" primes $\mathbf{m} = (\mathbf{m}_1, \ldots, \mathbf{m}_{\ell'})$ (except for the first element $\mathbf{m}_1 = 4$ that is not a prime). Initially, the parties will sample shares of candidates p and q that are $3 \pmod 4$ and in addition are not divisible by the first $\ell < \ell'$ elements. This is done by collectively sampling for every $j \in [\ell]$ values $\mathbf{p}_{i,j}, \mathbf{q}_{i,j} \in \mathbb{Z}_{\mathbf{m}_j}$ for every party $\mathcal{P}_i$, such that $\sum_i \mathbf{p}_{i,j} \cdot \sum_i \mathbf{q}_{i,j} \not\equiv 0 \pmod{\mathbf{m}_j}$.

The parties then "extend" the shares as follows: Each party $\mathcal{P}_i$ locally CRT-reconstructs its shares to get $p_i, q_i \in \mathbb{Z}_M$, where $M = \prod_{j \in [\ell]} \mathbf{m}_j$, and for every $j \in [\ell+1, \ell']$ computes $\mathbf{p}_{i,j} \equiv p_i \pmod{\mathbf{m}_j}$ and $\mathbf{q}_{i,j} \equiv q_i \pmod{\mathbf{m}_j}$. Next, to reconstruct the candidate biprime N, the parties jointly multiply their shares such that each $\mathcal{P}_i$ gets $\mathbf{N}_{i,j} \in \mathbb{Z}_{\mathbf{m}_j}$ that is a share of $\sum_i \mathbf{p}_{i,j} \cdot \sum_i \mathbf{q}_{i,j} \pmod{m_j}$. Each $\mathcal{P}_i$ broadcasts $\mathbf{N}_{i,j}$, and the parties locally CRT-reconstruct $N \in \mathbb{Z}_{M'}$, where $M' = \prod_{j \in [\ell']} \mathbf{m}_j$.

The first phase completes by having each party perform a local trial division on N.

Biprimality test: The parties jointly execute a biprimality test, where every party enters the candidate N and its shares p_i and q_i, and receives back a biprimality indicator. In the semi-honest case, the parties can complete the protocol at this point; however, in the malicious setting they need to check whether the adversary influenced the result by cheating.

Consistency check: In case the biprimality test returned that N is *not* a biprime, the parties reveal the shares they used during the protocol and verify that p and q are *not* both primes (indeed, in certain biprimality tests, e.g., [BF01], the adversary may have the capability to reject a valid biprime by cheating). If the biprimality test returned that N is a biprime, the parties run a test to ensure that N indeed corresponds to the shares used in the candidate-sieving phase (to make sure the adversary did not lie and used different shares in different parts of the protocol).

4.1 Ideal Functionalities used in the Protocol

4.1.1 Augmented Multiparty Multiplier

The augmented multiplier functionality $\mathcal{F}_{\mathsf{AugMul}}$ (Functionality 4.1) is a reactive functionality that operates in multiple phases and stores an internal state across calls. In its most basic form, the functionality allows the parties to:

- Sample shares of non-zero multiplication triplets over "small" primes. That is, given a prime m, the functionality receives a triplet $x_i, y_i, z_i \in \mathbb{Z}_m$ from every corrupted $\mathcal{P}_i$ and samples $x_i, y_i, z_i \leftarrow \mathbb{Z}_m$ for every honest $\mathcal{P}_i$, conditioned on $z \equiv x \cdot y \pmod{m}$ and $z \not\equiv 0 \pmod{m}$, where $x = \sum_i x_i$, $y = \sum_i y_i$, and $z = \sum_i z_i$. This allows the parties to sample candidates that are not divisible by "small" primes in CRT-form.

- Load arbitrary shares in $\mathbb{Z}_m$, for some prime m, into the memory of the functionality, and later multiply them to get shares of the product. That is, each $\mathcal{P}_i$ inputs $x_i, y_i \in \mathbb{Z}_m$ and receives back z_i such that $\sum z_i \equiv \sum x_i \cdot \sum y_i \pmod{m}$. This allows the parties to "extend" their shares from a CRT representation of an element in $\mathbb{Z}_M$, where $M = \prod_{j \in [\ell]} \mathbf{m}_j$, into a CRT representation of an element in $\mathbb{Z}_{M'}$, where $M' = \prod_{j \in [\ell']} \mathbf{m}_j$.

These interfaces are sufficient for the semi-honest version of the protocol template; however, as discussed above, in the malicious setting the parties additionally perform a consistency check to ensure that correct values have been used. Therefore, we add the following capabilities:

- Test a predicate over the set of stored values. Namely, if the biprimality test returned that N is a biprime, the parties will verify whether the shares stored by the functionality indeed CRT-reconstruct to the factors p and q of N. If not, the parties will learn that some parties deviated from the protocol.

- Reveal all the values stored in memory to all the parties. Namely, if the biprimality test returned that N is *not* a biprime, the parties will use this option to reveal all the shares and verify whether N is indeed not a biprime, or if some parties deviated from the protocol and caused a rejection sampling.

The interfaces described above suffice for the malicious version of the protocol template; however, to gain a substantial efficiency improvement in the protocol realizing $\mathcal{F}_{\mathsf{AugMul}}$, we weaken the guarantees offered by this functionality and allow the adversary more influence.

- Every value stored in the memory of the functionality is assigned with an sid. The adversary can ask the functionality to reveal values corresponding to sid's of its choice; however, in this case a "cheating flag" is set and the parties learn it when invoking the predicate-testing/input-revealing interfaces. Further, if the adversary is cheating during the sampling phase, it can determine the sampled values for the parties. Similarly, if the adversary is cheating during the multiplication phase, it can determine the value of the product's shares the parties receive, irrespectively of the actual values loaded by the parties.

Functionality 4.1. $\mathcal{F}_{\mathsf{AugMul}}(n)$. Augmented n-Party Multiplication

This functionality is parameterized by the party count n. In addition to the parties it interacts with an ideal adversary $\mathcal{S}$ who corrupts the parties indexed by $\mathbf{P}^*$. The remaining honest parties are indexed by $\overline{\mathbf{P}^*} := [n] \setminus \mathbf{P}^*$.

Cheater Activation: Upon receiving $(\texttt{cheat}, \mathsf{sid})$ from $\mathcal{S}$, store $(\texttt{cheater}, \mathsf{sid})$ in memory and send every record of the form $(\texttt{value}, \mathsf{sid}, i, x_i, m)$ to $\mathcal{S}$. For the purposes of this functionality, we will consider session IDs to be fresh even when a `cheater` record already exists in memory.

Sampling: Upon receiving $(\texttt{sample}, \mathsf{sid}_1, \mathsf{sid}_2, m)$ from each party $\mathcal{P}_i$ for $i \in \overline{\mathbf{P}^*}$ and $(\texttt{adv-sample}, \mathsf{sid}_1, \mathsf{sid}_2, x_i, y_i, z_i, m)$ from $\mathcal{S}$ for $i \in \mathbf{P}^*$,[a] if sid_1 and sid_2 are fresh, agreed-upon values and if m is an agreed-upon prime, and if neither $(\texttt{cheater}, \mathsf{sid}_1)$ nor $(\texttt{cheater}, \mathsf{sid}_2)$ exists in memory, then sample $(x_i, y_i, z_i) \leftarrow \mathbb{Z}_m^3$ uniformly for each $i \in \overline{\mathbf{P}^*}$ subject to

$$\sum_{i\in[n]} z_i \equiv \sum_{i\in[n]} x_i \cdot \sum_{i\in[n]} y_i \not\equiv 0 \pmod{m}$$

If the previous conditions hold, but $(\texttt{cheater}, \mathsf{sid}_1)$ or $(\texttt{cheater}, \mathsf{sid}_2)$ exists in memory, then send $(\texttt{cheat-sample}, \mathsf{sid}_1, \mathsf{sid}_2)$ to $\mathcal{S}$ and in response receive $(\texttt{cheat-product}, \mathsf{sid}_1, \mathsf{sid}_2, \{(x_i, y_i, z_i)\}_{i\in\overline{\mathbf{P}^*}})$ where $0 \le x_i, y_i, z_i < m$ for all i and where

$$\sum_{i\in[n]} z_i \not\equiv 0 \pmod{m}$$

(if these conditions are violated, then ignore the response from $\mathcal{S}$). Regardless, store $(\texttt{value}, \mathsf{sid}_1, i, x_i, m)$ and $(\texttt{value}, \mathsf{sid}_2, i, y_i, m)$ in memory for $i \in [n]$, and then send $(\texttt{sampled-value}, \mathsf{sid}_1, \mathsf{sid}_2, x_i, y_i, z_i)$ to each party $\mathcal{P}_i$ as adversarially delayed private output.

Input: Upon receiving $(\texttt{input}, \mathsf{sid}, x_i, m)$ from each party $\mathcal{P}_i$, where $i \in [n]$: if sid is a fresh, agreed-upon value and if m is an agreed-upon prime, and if $0 \le x_i < m$ for all $i \in [n]$, then store $(\texttt{value}, \mathsf{sid}, i, x_i, m)$ in memory for each $i \in [n]$ and send $(\texttt{value-loaded}, \mathsf{sid})$ to all parties. If $(\texttt{cheater}, \mathsf{sid})$ exists in memory, then send $(\texttt{value}, \mathsf{sid}, i, x_i, m)$ to $\mathcal{S}$ for each $i \in [n]$.

Multiplication: Upon receiving $(\texttt{multiply}, \mathsf{sid}_1, \mathsf{sid}_2, \mathsf{sid}_3)$ from each party $\mathcal{P}_i$ for $i \in \overline{\mathbf{P}^*}$ and $(\texttt{adv-multiply}, \mathsf{sid}_1, \mathsf{sid}_2, \mathsf{sid}_3, i, z_i)$ from $\mathcal{S}$ for each $i \in \mathbf{P}^*$,[a] if all three session IDs are agreed upon and sid_3 is fresh, and if no record of the form $(\texttt{cheater}, \mathsf{sid}_1)$ or $(\texttt{cheater}, \mathsf{sid}_2)$ exists in memory, and if records of the form $(\texttt{value}, \mathsf{sid}_1, i, x_i, m_1)$ and $(\texttt{value}, \mathsf{sid}_2, i, y_i, m_2)$ exist in memory for all $i \in [n]$ such that $m_1 = m_2$, then sample $z_i \leftarrow \mathbb{Z}_{m_1}$ for $i \in \overline{\mathbf{P}^*}$ subject to

$$\sum_{i\in[n]} z_i \equiv \sum_{i\in[n]} x_i \cdot \sum_{i\in[n]} y_i \pmod{m_1}$$

If the previous conditions hold, but $(\texttt{cheater}, \mathsf{sid}_1)$ or $(\texttt{cheater}, \mathsf{sid}_2)$ exists in memory, then send $(\texttt{cheat-multiply}, \mathsf{sid}_1, \mathsf{sid}_2, \mathsf{sid}_3)$ to $\mathcal{S}$ and in response receive $(\texttt{cheat-product}, \mathsf{sid}_3, \{z_i\}_{i\in\overline{\mathbf{P}^*}})$ where $0 \le z_i < m_1$ for all i. Regardless, send $(\texttt{product}, \mathsf{sid}_3, z_i)$ to each party $\mathcal{P}_i$ for $i \in [n]$ as adversarially delayed private output. Note that this procedure only permits multiplications of values associated with the *same* modulus.

Predicate Cheater Check: Upon receiving $(\texttt{check}, \mathsf{sids}, f)$ from all parties, where f is the description of a predicate over the set of stored values associated with the vector of session IDs sids, if f is not agreed upon, or if any record $(\texttt{cheater}, \mathsf{sid})$ exists in memory such that $\mathsf{sid} \in \mathsf{sids}$, then abort, informing all parties in an adversarially delayed fashion. Otherwise, let $\mathbf{x}$ be the vector of stored values associated with sids, or in other words, let it be a vector such that for all $j \in [|\mathbf{x}|]$ and $i \in [n]$, records of the form $(\texttt{value}, \mathsf{sids}_j, i, y_i, m)$ exist in memory such that

$$0 \le \mathbf{x}_j < m \qquad \text{and} \qquad \mathbf{x}_j \equiv \sum_{i\in[n]} y_i \pmod{m}$$

Send $(\texttt{predicate-result}, \mathsf{sids}, f(\mathbf{x}))$ to all parties as adversarially delayed private output, and refuse all future messages with any session ID in sids.

Input Revelation: Upon receiving $(\texttt{open}, \mathsf{sid})$ from all parties, if a record of the form $(\texttt{cheater}, \mathsf{sid})$ exists in memory, then abort, informing all parties in an adversarially delayed fashion. Otherwise, for each record of the form $(\texttt{value}, \mathsf{sid}, i, x_i)$ in memory, send $(\texttt{opening}, \mathsf{sid}, i, x_i)$ to all parties as adversarially delayed output. Refuse all future messages with this sid.

[a]In the semi-honest setting, the adversary does not send these values to the functionality; instead the functionality samples the shares for corrupt parties just as it does for honest parties.

4.1.2 Biprimality Test

The biprimality-test functionality $\mathcal{F}_{\mathsf{Biprime}}$ (Functionality 4.2) abstracts the behavior of the biprimality test of Boneh and Franklin [BF01]. The functionality receives from each party the candidate N along with its shares of p and of q, and checks whether p and q are primes and if $N = p \cdot q$. The adversary is given the capability to cheat and learn *all* the shares of p and of q, but in this case the functionality returns that N is not a biprime (even if p and q are in fact primes).

Functionality 4.2. $\mathcal{F}_{\mathsf{Biprime}}(n, M)$. Distributed Biprimality Test

This functionality is parameterized by the integer M and the number of parties n. In addition to the parties it interacts with an ideal adversary $\mathcal{S}$.

Biprimality Test:

1. Wait to receive $(\texttt{check-biprimality}, \mathsf{sid}, N, p_i, q_i)$ from each party $\mathcal{P}_i$ for $i \in [n]$, where sid is a fresh, agreed-upon value.
2. Over the integers, compute
$$p := \sum_{i\in[n]} p_i \qquad \text{and} \qquad q := \sum_{i\in[n]} q_i \qquad \text{and} \qquad N' := p \cdot q$$
3. If all parties agreed on the value of N in Step 1, and $N = N'$, and both p and q are primes, and $p \not\equiv 1 \pmod{q}$, and $q \not\equiv 1 \pmod{p}$, and $0 \leq p < M$ and $0 \leq q < M$, and $\mathcal{S}$ has *not* previously sent a message $(\texttt{leak-shares}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{Biprime}}$, then output $(\texttt{biprime}, \mathsf{sid})$ to all parties as adversarially delayed output. Otherwise, output $(\texttt{leaked-shares}, \mathsf{sid}, \{(p_i, q_i)\}_{i\in[n]})$ directly to $\mathcal{S}$, and output $(\texttt{not-biprime}, \mathsf{sid})$ to all parties as adversarially delayed output.

4.2 The Protocol Template

Having defined the functionalities $\mathcal{F}_{\mathsf{AugMul}}$ and $\mathcal{F}_{\mathsf{Biprime}}$, we are now ready to present the distributed RSA protocol. Our protocol requires the existence of a set of primes $\mathbf{m} = (\mathbf{m}_1, \ldots, \mathbf{m}_{\ell'})$, defined relative to the security parameter κ and the number of parties n, such that the first ℓ elements $(\mathbf{m}_1, \ldots, \mathbf{m}_\ell)$ (used for the CRT-based sampling) are not "too large" and that remaining elements $(\mathbf{m}_{\ell+1}, \ldots, \mathbf{m}_{\ell'})$ (used for the "share extension") are not "too small." We formalize this as follows.

Definition 4.3 (Compatible Parameter)**.** *Let κ, n, ℓ, and ℓ' be integers, and let $\mathbf{m} = (\mathbf{m}_1, \ldots, \mathbf{m}_{\ell'})$ be a vector beginning with 4, and then containing the first $\ell' - 1$ odd primes in ascending order. We say that $\mathbf{m}$ is a $(\kappa, n, \ell, \ell', M)$-*compatible parameter *if (a) $\prod_{j\in[\ell']} \mathbf{m}_j > 2^{2\kappa}$, (b) the first ℓ elements of $\mathbf{m}$ are the (κ, n)-near-primorial vector (per Definition 3.2), and (c) $M = \prod_{j\in[\ell]} \mathbf{m}_j$.*

Note that the efficiency of our protocol relies on the choice of $\mathbf{m}$ because we use these sets to sieve candidate primes. Since smaller numbers are more likely to be factors for the candidate primes, we choose $\mathbf{m}$ as the largest allowable set of the smallest sequential primes.

Protocol 4.4. $\pi_{\mathsf{RSAGen}}(\kappa, n, B)$**. Distributed Biprime Sampling**

This protocol is parameterized by the RSA security parameter κ, the number of parties n, and the trial-division bound B. Let $\mathbf{m}$ be a $(\kappa, n, \ell, \ell', M)$-compatible parameter. The parties participating in this protocol have access to the functionalities $\mathcal{F}_{\mathsf{AugMul}}(n)$ and $\mathcal{F}_{\mathsf{Biprime}}(n, M)$.

Candidate Sieving:

1. Upon receiving input $(\texttt{sample}, \mathsf{sid})$, every party $\mathcal{P}_i$ computes three vectors of Session IDs

$$\begin{aligned} \mathbf{psids} &:= \{\mathsf{GenSID}(\mathsf{sid}, j, \mathsf{p})\}_{j\in[\ell']} \\ \mathbf{qsids} &:= \{\mathsf{GenSID}(\mathsf{sid}, j, \mathsf{q})\}_{j\in[\ell']} \\ \mathbf{Nsids} &:= \{\mathsf{GenSID}(\mathsf{sid}, j, \texttt{N})\}_{j\in[\ell']} \end{aligned}$$

   sends $(\texttt{sample}, \mathbf{psids}_j, \mathbf{qsids}_j, \mathbf{m}_j)$ to $\mathcal{F}_{\mathsf{AugMul}}(n)$ for every $j \in [2, \ell]$, and receives $(\texttt{sampled-product}, \mathbf{psids}_j, \mathbf{qsids}_j, \mathbf{p}_{i,j}, \mathbf{q}_{i,j}, \mathbf{N}_{i,j})$ in response. Party $\mathcal{P}_1$ also sets $\mathbf{p}_{1,1} := \mathbf{q}_{1,1} := 3$, and all others set $\mathbf{p}_{i,1} := \mathbf{q}_{i,1} := 0$.

2. Each party $\mathcal{P}_i$ for $i \in [n]$ computes

$$\begin{aligned} p_i &:= \mathsf{CRTRecon}\left(\{\mathbf{m}_j\}_{j\in[\ell]}, \{\mathbf{p}_{i,j}\}_{j\in[\ell]}\right) \\ q_i &:= \mathsf{CRTRecon}\left(\{\mathbf{m}_j\}_{j\in[\ell]}, \{\mathbf{q}_{i,j}\}_{j\in[\ell]}\right) \end{aligned}$$

   and then, for $j \in [\ell + 1, \ell']$, $\mathcal{P}_i$ computes

$$\mathbf{p}_{i,j} := p_i \bmod \mathbf{m}_j \qquad \text{and} \qquad \mathbf{q}_{i,j} := q_i \bmod \mathbf{m}_j$$

   Note that each party $\mathcal{P}_i$ is now in possession of a pair of vectors

$$\mathbf{p}_{i,*} \in \mathbb{Z}_{\mathbf{m}_1} \times \ldots \times \mathbb{Z}_{\mathbf{m}_{\ell'}} \qquad \text{and} \qquad \mathbf{q}_{i,*} \in \mathbb{Z}_{\mathbf{m}_1} \times \ldots \times \mathbb{Z}_{\mathbf{m}_{\ell'}}$$

3. For $j \in [\ell+1, \ell']$, every party $\mathcal{P}_i$ for $i \in [n]$ sends the following sequence of messages to $\mathcal{F}_{\mathsf{AugMul}}(n)$, waiting for confirmation after each:

   (a) $(\texttt{input}, \mathsf{psids}_j, \mathbf{p}_{i,j}, \mathbf{m}_j)$

   (b) $(\texttt{input}, \mathsf{qsids}_j, \mathbf{q}_{i,j}, \mathbf{m}_j)$

   (c) $(\texttt{multiply}, \mathsf{psids}_j, \mathsf{qsids}_j, \mathsf{Nsids}_j)$

   and at the end of this sequence, each party $\mathcal{P}_i$ receives $(\texttt{product}, \mathsf{Nsids}_j, \mathbf{N}_{i,j})$ from $\mathcal{F}_{\mathsf{AugMul}}(n)$ in response. Note that each party $\mathcal{P}_i$ is now in possession of a vector $\mathbf{N}_{i,*} \in \mathbb{Z}_{\mathbf{m}_1} \times \ldots \times \mathbb{Z}_{\mathbf{m}_{\ell'}}$.

4. For $j \in [\ell']$, each party $\mathcal{P}_i$ for $i \in [n]$ broadcasts $\mathbf{N}_{i,j}$. Once all parties have received shares from all other parties, they compute

$$N := \sum_{i' \in [n]} \mathsf{CRTRecon}\left(\mathbf{m}, \mathbf{N}_{i',*}\right)$$

5. Each party $\mathcal{P}_i$ performs a local trial division on N by all primes less than B. If N is divisible by some prime, then the parties skip directly to Step 7, and take the privacy-free branch.

Biprimality Test:

6. Each party $\mathcal{P}_i$ for $i \in [n]$ sends $(\texttt{check-biprimality}, \mathsf{sid}, N, p_i, q_i)$ to $\mathcal{F}_{\mathsf{Biprime}}(n, M)$ and waits for either $(\texttt{biprime}, \mathsf{sid})$ or $(\texttt{not-biprime}, \mathsf{sid})$ in response.

Consistency Check: [a]

7. Let f be the predicate that is defined to return 1 if and only if

$$N = \mathsf{CRTRecon}\left(\mathbf{m}, \sum_{i' \in [n]} \mathbf{p}_{i',*}\right) \cdot \mathsf{CRTRecon}\left(\mathbf{m}, \sum_{i' \in [n]} \mathbf{q}_{i',*}\right)$$

   where the sums are taken over the integers.

   - If $\texttt{biprime}$ is received from $\mathcal{F}_{\mathsf{Biprime}}(n, M)$, then N is a biprime, and a privacy-preserving check must be performed. Each party sends $(\texttt{check}, \mathsf{psids} \| \mathsf{qsids}, f)$ to $\mathcal{F}_{\mathsf{AugMul}}(n)$. If $\mathcal{F}_{\mathsf{AugMul}}$ returns $(\texttt{predicate-result}, \mathsf{psids} \| \mathsf{qsids}, 1)$ then halt successfully with output $(\texttt{biprime}, \mathsf{sid}, N)$; otherwise, abort.
   - If $\texttt{not-biprime}$ is received from $\mathcal{F}_{\mathsf{Biprime}}(n, M)$, then either N is not a biprime or some party has cheated; consequently, a privacy-free check is performed.

(a) For $j \in [2, \ell']$, each party $\mathcal{P}_i$ for $i \in [n]$ sends $(\texttt{open}, \mathsf{psids}_j)$ and $(\texttt{open}, \mathsf{qsids}_j)$ to $\mathcal{F}_{\mathsf{AugMul}}(n)$. If $\mathcal{P}_i$ observes $\mathcal{F}_{\mathsf{AugMul}}(n)$ to abort in response to any of these queries, then $\mathcal{P}_i$ itself aborts. Otherwise, $\mathcal{P}_i$ receives $(\texttt{opening}, \mathsf{psids}_j, \mathbf{p}_{i',j})$ and $(\texttt{opening}, \mathsf{qsids}_j, \mathbf{q}_{i',j})$ for each $i' \in [n]$ and $j \in [2, \ell']$.

(b) The parties individually check that the predicate f holds over the vectors of shares which they now all possess. If this predicate holds and p and q are not both prime, then all parties halt successfully with output $(\texttt{non-biprime}, \mathsf{sid})$. Otherwise, a party has cheated, and they abort.

[a]If only security against semi-honest adversaries is required, the protocol can terminate after the Biprimality-Test phase, and these checks are unnecessary.

4.3 Security Proof

We proceed to show that the protocol template π_{RSAGen} realizes the functionality $\mathcal{F}_{\mathsf{RSAGen}}$. In the semi-honest setting this holds unconditionally facing computationally unbounded adversaries with perfect correctness. In the malicious setting this holds against PPT adversaries under the assumption that factoring is hard with respect to BFGM.

Theorem 4.5. *Protocol π_{RSAGen} UC-realizes $\mathcal{F}_{\mathsf{RSAGen}}$ with perfect security in the $(\mathcal{F}_{\mathsf{AugMul}}, \mathcal{F}_{\mathsf{Biprime}})$-hybrid model against a static, semi-honest adversary that corrupts up to $n-1$ parties.*

Proof. Let $\mathcal{A}$ be a semi-honest adversary. We will construct a simulator $\mathcal{S}$ such that no environment can distinguish between running with $\mathcal{A}$ and real parties executing π_{RSAGen} and running with $\mathcal{S}$ and dummy parties that interact with $\mathcal{F}_{\mathsf{RSAGen}}$. Let $\mathcal{Z}$ be an environment.

The simulator forwards any message it receives from $\mathcal{Z}$ to $\mathcal{A}$ and vice-versa. To simulate Step 1, $\mathcal{S}$ receives $(\texttt{sample}, \mathsf{psids}_j, \mathsf{qsids}_j, \mathbf{m}_j)$ from $\mathcal{A}$ for every $j \in [2, \ell]$ and on behalf of every corrupted party $\mathcal{P}_i$, samples uniformly random values $\mathbf{p}_{i,j}, \mathbf{q}_{i,j}, \mathbf{N}_{i,j} \leftarrow \mathbb{Z}_{\mathbf{m}_j}$ and answers $(\texttt{sampled-product}, \mathsf{psids}_j, \mathsf{qsids}_j, \mathbf{p}_{i,j}, \mathbf{q}_{i,j}, \mathbf{N}_{i,j})$.

$\mathcal{S}$ reconstructs the shares of each corrupted $\mathcal{P}_i$ by computing $p_i := \mathsf{CRTRecon}(\{\mathbf{m}_j\}_{j\in[\ell]}, \{\mathbf{p}_{i,j}\}_{j\in[\ell]})$ and $q_i := \mathsf{CRTRecon}(\{\mathbf{m}_j\}_{j\in[\ell]}, \{\mathbf{q}_{i,j}\}_{j\in[\ell]})$, and sends $(\texttt{adv-input}, \mathsf{sid}, i, p_i, q_i)$ to $\mathcal{F}_{\mathsf{RSAGen}}$. Next, $\mathcal{S}$ receives from $\mathcal{F}_{\mathsf{RSAGen}}$ either $(\texttt{biprime}, \mathsf{sid}, N)$ or $(\texttt{non-biprime}, \mathsf{sid}, p, q)$ (in the latter case $\mathcal{S}$ computes $N = p \cdot q$) and responds with $(\texttt{proceed}, \mathsf{sid})$.

To simulate Step 3, $\mathcal{S}$ receives (sequentially) $(\texttt{input}, \mathsf{psids}_j, \mathbf{p}_{i,j}, \mathbf{m}_j)$ followed by $(\texttt{input}, \mathsf{qsids}_j, \mathbf{q}_{i,j}, \mathbf{m}_j)$ and $(\texttt{multiply}, \mathsf{psids}_j, \mathsf{qsids}_j, \mathsf{Nsids}_j)$ from $\mathcal{A}$ on behalf of every corrupted party $\mathcal{P}_i$ and for every $j \in [\ell+1, \ell']$ and answers accordingly with $(\texttt{value-loaded}, \mathsf{sid})$ as $\mathcal{F}_{\mathsf{AugMul}}$ would, where in the last message it replies with $(\texttt{product}, \mathsf{Nsids}_j, \mathbf{N}_{i,j})$ for a uniformly distributed $\mathbf{N}_{i,j} \in \mathbb{Z}_{\mathbf{m}_j}$.

To simulate Step 4, for every $j \in [\ell']$ the simulator $\mathcal{S}$ samples uniformly distributed values $\mathbf{N}_{i,j} \in \mathbb{Z}_{\mathbf{m}_j}$ for every honest party $\mathcal{P}_i$ conditioned on $N \equiv$

$\sum_i \mathbf{N}_{i,j} \pmod{\mathbf{m}_j}$, and simulates broadcasting $\mathbf{N}_{i,j}$ to $\mathcal{A}$ on behalf of the honest $\mathcal{P}_i$ (recall that the values $\mathbf{N}_{i,j}$ have already been set for corrupted parties). In addition, $\mathcal{S}$ receives $\mathbf{N}_{i,j}$ from $\mathcal{A}$ on behalf of every corrupted $\mathcal{P}_i$.

To simulate Step 5, $\mathcal{S}$ tries to factor N by all primes smaller than B. If N is factored, the simulation of the protocol completes. Otherwise, to simulate Step 6, $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{Biprime}}$ by receiving $(\texttt{check-biprimality}, \mathsf{sid}, N, p_i, q_i)$ from $\mathcal{A}$ on behalf of every corrupted party $\mathcal{P}_i$ and responds with $(\texttt{biprime}, \mathsf{sid})$ or $(\texttt{not-biprime}, \mathsf{sid})$ accordingly.

It is immediate to verify that the view of any environment is identically distributed when running with $\mathcal{A}$ and real parties executing π_{RSAGen} and running with $\mathcal{S}$ and dummy parties that interact with $\mathcal{F}_{\mathsf{RSAGen}}$. This concludes the proof of Theorem 4.5. □

We proceed to prove the security of π_{RSAGen} in the malicious setting. We provide a proof sketch below; see Appendix D for the full proof.

Theorem 4.6. *Assume that factoring is hard with respect to* BFGM*. Then, protocol* π_{RSAGen} *UC-realizes* $\mathcal{F}_{\mathsf{RSAGen}}$ *in the* $(\mathcal{F}_{\mathsf{AugMul}}, \mathcal{F}_{\mathsf{Biprime}})$*-hybrid model against a static, malicious adversary that corrupts up to* $n-1$ *parties.*

Proof Sketch. We observe that if the adversary simply follows the specification of the protocol and does not cheat in $\mathcal{F}_{\mathsf{AugMul}}$ or $\mathcal{F}_{\mathsf{Biprime}}$, the simulation follows similarly to the semi-honest case. At any point if the adversary were to deviate from the protocol, the simulator will request $\mathcal{F}_{\mathsf{RSAGen}}$ to reveal all honest parties' shares and makes appropriate use of them in subsequent interactions with the adversary, effectively running the code of honest parties. This matches the adversary's view in the real protocol as far as the distribution of the honest parties' shares is concerned.

It remains to be argued that any deviation from the protocol specification will also result in an abort in the real world with honest parties, and will additionally be recognized by the honest parties as an adversarially induced cheat (as opposed to a statistical sampling failure). Note that detecting cheats is only needed when N is a biprime and the adversary has sabotaged a successful candidate, and not the instances where N is not a biprime, hence was going to be rejected anyway.

We analyze all possible cases where the adversary deviates from the protocol below. Let N be the value set by parties' values in the candidate-sieving phase.

Case 1: N is a non-biprime and reconstructed correctly. In this case, $\mathcal{F}_{\mathsf{Biprime}}$ will always reject N as there exist no satisfying inputs (i.e., there are no two prime factors p, q such that $p \cdot q = N$).

Case 2: N is a non-biprime and reconstructed incorrectly as N'. If by fluke N' happens to be a biprime, this incorrect reconstruction will be caught during the consistency-check phase due to the explicit predicate check in $\mathcal{F}_{\mathsf{AugMul}}$. Otherwise, if N' is a non-biprime, the same argument from the previous case applies.

Case 3: N is a biprime and reconstructed correctly. If consistent inputs are used for the biprimality test and nobody cheats, the candidate N is successfully accepted (this case essentially corresponds to the semi-honest case). Otherwise, if inconsistent inputs are used for the biprimality test, one of the following options will happen:

- $\mathcal{F}_{\mathsf{Biprime}}$ rejects this candidate. In this case, all parties reveal their inputs shares (guaranteed correct via $\mathcal{F}_{\mathsf{AugMul}}$) publicly reconstruct p and q and locally test their primality. This will verify that N was a biprime, and that $\mathcal{F}_{\mathsf{Biprime}}$ must have been supplied with inconsistent inputs, implying that some party has cheated.

- $\mathcal{F}_{\mathsf{Biprime}}$ accepts this candidate. This case occurs with negligible probability (assuming factoring is hard), for the following reason. As N only has two factors, there is exactly one *other* pair of inputs that the adversary can supply to $\mathcal{F}_{\mathsf{Biprime}}$ to induce this scenario (besides the one specified by the protocol). We present a reduction from finding this alternative satisfying input to solving the factoring problem. We are careful to rely on the hardness of factoring only in this case, where by premise the N generated is a biprime with κ-bit factors, i.e., an instance of the factoring problem.

Case 4: N is a biprime and reconstructed incorrectly as N'. If N' is a biprime, this incorrect reconstruction will be caught during the consistency-check phase, just as when N is a biprime. If N' is a non-biprime, it will by rejected by $\mathcal{F}_{\mathsf{Biprime}}$, inducing all parties to reveal their shares and find that their shares do not in fact reconstruct to N', meaning that some party has cheated.

Thus the adversary is always caught when trying to sabotage a biprime candidate, and can never sneak a non-biprime past the consistency check. □

5 Distributed Biprimality Testing

5.1 The Semi-Honest Setting

Our protocol is based on the technique of Boneh and Franklin [BF01]. Loosely speaking, the Boneh-Franklin protocol works by utilizing a variant of the Miller-Rabin test: for a randomly chosen $\gamma \in \mathbb{Z}_N^*$ with Jacobi symbol 1, test whether $\gamma^{(N-p-q+1)/4} \equiv \pm 1 \pmod{N}$. A biprime will always pass this test, but non-biprimes may yield a false positive with probability 1/2. In order to bound the probability of proceeding with a false positive by 2^{-s} (where s is a statistical parameter), the test is repeated s times.

The above test does not filter out the remaining class of non-biprimes of the form $p = r_1^{d_1}$, $q = r_2^{d_2}$, and $q \equiv 1 \pmod{r_1^{d_1-1}}$. These non-biprimes are handled by the following procedure: sample $r \leftarrow \mathbb{Z}_N$, compute $z = r \cdot (p+q-1)$ and test whether $\gcd(z, N) = 1$. We express the computation of z as a circuit and make use of generic MPC (modeled via the $\mathcal{F}_{\mathsf{ComCompute}}$ functionality; see

Appendix B.2) to compute it. This test identifies every non-biprime fitting the above description, while inducing a negligibly small false negative rate. We refer the reader to [BF01, Section 4.1] for a more comprehensive discussion. The following lemma follows immediately from [BF01].

Lemma 5.1. *The protocol described in [BF01] UC-realizes $\mathcal{F}_{\mathsf{Biprime}}$ in the $\mathcal{F}_{\mathsf{ComCompute}}$-hybrid model against a static, semi-honest adversary who corrupts up to $n-1$ parties.*

5.2 The Malicious Setting

Compared to the semi-honest setting, a malicious adversary has the ability to make a biprime N fail the test; however, a non-biprime cannot pass the test. Our maliciously-secure protocol is inspired by [FLOP18], but we make many simplifications and prove that it UC-realizes our $\mathcal{F}_{\mathsf{Biprime}}$ functionality.

The first step of our protocol is for all parties to commit to their shares of p and q, along with a random pad τ. The semi-honest Boneh-Franklin biprimality test is then run (except for the gcd-checking component), and a consistency check is used to prove that $\chi = \gamma^{-(p+q-1)/4}$ was computed correctly. In particular once χ is public, parties broadcast $\alpha = \gamma^{\tau}$, sample a random challenge bit $c \leftarrow \{0,1\}$, and reveal $\zeta = \tau - c \cdot (p+q-1)/4$. Verifying $\gamma^{\zeta} \overset{?}{=} \alpha \cdot \chi^{c}$ proves that χ was constructed correctly with a soundness error of $1/2$. The semi-honest Boneh-Franklin test is augmented with this check to achieve an overall statistical soundness error of $3/4$, and is repeated sufficiently many times to achieve the desired soundness error of 2^{-s}.

In the above protocol, the correct relationship between ζ, τ, p, and q is verified via generic MPC (modeled via $\mathcal{F}_{\mathsf{ComCompute}}$). Simultaneously, the gcd check is run within the MPC to sample and output $z = r \cdot (p+q-1)$. Our protocol is defined in a hybrid model with the coin-tossing functionality $\mathcal{F}_{\mathsf{CT}}$, the one-to-many commitment functionality $\mathcal{F}_{\mathsf{Com}}$, and the integer-sharing of zero functionality $\mathcal{F}_{\mathsf{Zero}}$; all functionalities are defined in Appendix B.2. In addition, the protocol uses the algorithm $\mathsf{VerifyBiprime}$ defined below (Algorithm 5.3).

Protocol 5.2. $\pi_{\mathsf{biprime}}(n, M)$. **Distributed Biprimality Testing**

This protocol is parameterized by the number of parties n, the statistical security parameter s, and an integer M. The parties have access to the $\mathcal{F}_{\mathsf{CT}}$, $\mathcal{F}_{\mathsf{Com}}$, $\mathcal{F}_{\mathsf{ComCompute}}$, and $\mathcal{F}_{\mathsf{Zero}}$ functionalities. This description assumes the perspective of party $\mathcal{P}_i$.

Commitment to inputs:

1. Upon receiving input $(\texttt{check-biprimality}, \mathsf{sid}, N, p_i, q_i)$:
   (a) Sample $\boldsymbol{\tau}_{j,i} \leftarrow \mathbb{Z}^{s}_{M \cdot 2^{2s+1}}$ for each $j \in [3s]$.
   (b) Send $(\texttt{commit}, \mathsf{sid}_i, (p_i, q_i, \boldsymbol{\tau}_{*,i}))$ to $\mathcal{F}_{\mathsf{ComCompute}}$.

Boneh-Franklin Test:

2. Send $(\texttt{sample}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{Zero}}(2^{2\kappa+s})$ and receive $(\texttt{zero-share}, \mathsf{sid}, r_i)$ in response.

3. Send $(\texttt{flip}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{CT}}$ and receive $(\texttt{coin}, \mathsf{sid}, x)$ in response. Use the random coins x to sample a vector $\boldsymbol{\gamma} \leftarrow (\mathbb{Z}_N^*)^{3s}$ such that the Jacobi symbol $J(\boldsymbol{\gamma}_j) = 1$ for all $j \in [3s]$.

4. For every $j \in [3s]$,

   (a) For $i = 1$, compute $\boldsymbol{\chi}_{j,i} := \boldsymbol{\gamma}_j^{r_i-(p_i+q_i-6)/4} \mod N$, and for $i > 1$, compute $\boldsymbol{\chi}_{j,i} := \boldsymbol{\gamma}_j^{r_i-(p_i+q_i)/4} \mod N$.
   Send $(\texttt{commit}, \mathsf{sid}_{j,i}, \boldsymbol{\chi}_{j,i}, [n])$ to $\mathcal{F}_{\mathsf{Com}}$.

   (b) On receiving $(\texttt{committed}, \mathsf{sid}_{j,i'})$ from $\mathcal{F}_{\mathsf{Com}}$ for every $i' \in [n]$, send $(\texttt{decommit}, \mathsf{sid}_{j,i})$ to $\mathcal{F}_{\mathsf{Com}}$.

   (c) Wait to receive $(\texttt{decommitted}, \mathsf{sid}_{j,i'}, \boldsymbol{\chi}_{j,i'})$ from $\mathcal{F}_{\mathsf{Com}}$ for all $i' \in [n]$. If
   $$\boldsymbol{\gamma}_j^{(N-5)/4} \cdot \prod_{i\in[n]} \boldsymbol{\chi}_{j,i} \not\equiv \pm 1 \pmod N$$
   then output $(\texttt{not-biprime}, \mathsf{sid})$ and halt.

Consistency Check and gcd **Test:**

5. For each $j \in [3s]$:

   (a) Compute $\boldsymbol{\alpha}_{j,i} = \boldsymbol{\gamma}_j^{\boldsymbol{\tau}_{j,i}} \mod N$ and broadcast $\boldsymbol{\alpha}_{j,i}$ to all other parties.

   (b) Send $(\texttt{flip}, \mathsf{sid}_j)$ $\mathcal{F}_{\mathsf{CT}}$ to obtain a random bit $\mathbf{c}_j \in \{0, 1\}$.

   (c) For $i = 1$ compute $\boldsymbol{\zeta}_{j,i} = \boldsymbol{\tau}_{j,i} - \mathbf{c}_j(p_i+q_i)/4$, and for $i > 1$, compute $\boldsymbol{\zeta}_{j,i} = \boldsymbol{\tau}_{j,i} - \mathbf{c}_j(p_i + q_i - 6)/4$.
   Broadcast $\boldsymbol{\zeta}_{j,i}$ to all other parties, and receive $\boldsymbol{\zeta}_{j,i'}$ for every $i' \in [n]$.

   (d) Abort if
   $$\prod_{i'\in[n]} \boldsymbol{\gamma}_j^{\boldsymbol{\zeta}_{j,i'}} \not\equiv \prod_{i'\in[n]} \boldsymbol{\alpha}_{j,i'} \cdot \boldsymbol{\chi}_{j,i'}^{\mathbf{c}_j} \pmod N$$

6. Let $C := C(\mathsf{VerifyBiprime}(N, M, \mathbf{c}, \{\cdot, \cdot, \cdot, \boldsymbol{\zeta}_{*,i'}\}_{i'\in[n]}))$ be the canonical circuit representation of Algorithm 5.3, with the public values N, M, $\mathbf{c}$, and $\boldsymbol{\zeta}$ hardcoded into the circuit. Send $(\texttt{compute}, \mathsf{sid}, \{\mathsf{sid}_{i'}\}_{i'\in[n]}, C)$ to $\mathcal{F}_{\mathsf{ComCompute}}$, and receive in response $(\texttt{result}, \mathsf{sid}, z)$. If $z = \perp$, abort.

7. If $\gcd(z, N) = 1$, then terminate with output $(\texttt{biprime}, \mathsf{sid})$; otherwise, with output $(\texttt{not-biprime}, \mathsf{sid})$.

Below we present the algorithm VerifyBiprime that is used for the gcd-based test.

Algorithm 5.3. VerifyBiprime$(N, M, \mathbf{c}, \{(p_i, q_i, \boldsymbol{\tau}_{*,i}, \boldsymbol{\zeta}_{*,i})\}_{i\in[n]})$

1. Sample $r \leftarrow \mathbb{Z}_N$ and compute

$$z := r \cdot \Big(-1 + \sum_{i\in[n]} (p_i + q_i) \Big) \bmod N$$

2. Return z if and only if all of the following predicates hold:

   (a) $\Big(\sum_{i\in[n]} p_i\Big) \cdot \Big(\sum_{i\in[n]} q_i\Big) \overset{?}{=} N$

   (b) $p_i, q_i < M$ for all $i \in [n]$

   (c) $\boldsymbol{\tau}_{j,1} = \boldsymbol{\zeta}_{j,1} + \mathbf{c}_j \cdot (p_1 + q_1 - 6)/4$ for each $j \in [3s]$

   (d) $\boldsymbol{\tau}_{j,i} = \boldsymbol{\zeta}_{j,i} + \mathbf{c}_j \cdot (p_i + q_i)/4$ for all $i \in [2, n]$, $j \in [3s]$

   If any of them do not hold, output $\perp$.

We proceed to prove the security of $\mathcal{F}_{\mathsf{Biprime}}$ in the malicious setting.

Theorem 5.4. *The protocol π_{biprime} UC-realizes $\mathcal{F}_{\mathsf{Biprime}}$ in the $(\mathcal{F}_{\mathsf{Com}}, \mathcal{F}_{\mathsf{ComCompute}}, \mathcal{F}_{\mathsf{CT}}, \mathcal{F}_{\mathsf{Zero}})$-hybrid model against a static, malicious adversary that corrupts up to $n-1$ parties.*

Proof Sketch. The simulation of instances where the adversary does not deviate from the protocol follows trivially from Boneh and Franklin [BF01]. We sketch below why security holds against malicious adversaries; in particular, we examine the case where N is not a biprime, and argue that even a computationally unbounded malicious adversary is unable to convince an honest party to output `biprime` except with probability negligible in s.

Let $p_{\mathcal{A}} = \sum_{i\in\mathbf{P}^*} p_i$ and $q_{\mathcal{A}} = \sum_{i\in\mathbf{P}^*} q_i$ as per the adversarial inputs to $\mathcal{F}_{\mathsf{ComCompute}}$, and let $\tau_j = \sum_i \boldsymbol{\tau}_{j,i}$. First, note that $\tau_j + p + q - 1$ statistically hides p and q since $|\tau_j|/|p| > 2^s$ and τ_j is chosen uniformly. Assume that the tests made in Step 2 of VerifyBiprime are satisfied. We trace the outcome of each iteration of the Boneh-Franklin test (i.e., Step 4 of π_{biprime}) and examine the adversary's influence.

When $q \not\equiv 1 \pmod p$, false positives can be triggered either as a statistical event with probability 1/2, when $\boldsymbol{\gamma}_j^{(N-p-q)/4} \equiv \pm 1 \pmod N$ (as discussed in the semi-honest case), or adversarially when $\boldsymbol{\gamma}_j^{(N-p-q)/4} \not\equiv \pm 1 \pmod N$. Define $\Delta_{1,j}, \Delta_{2,j} \in \mathbb{Z}_{\varphi(N)}$, for $j \in [3s]$, such that

$$\prod_{i\in[n]} \boldsymbol{\chi}_{j,i} \equiv \boldsymbol{\gamma}_j^{-(p+q-6+\Delta_{1,j})} \pmod N \quad \text{and} \quad \prod_{i\in[n]} \boldsymbol{\alpha}_{j,i} \equiv \boldsymbol{\gamma}_j^{\tau_j + \Delta_{2,j}} \pmod N$$

Consider a non-biprime N. A simple calculation shows that in the event that the adversary is able to influence the j'th iteration to accept N, it holds that $\Delta_{2,j} \equiv \mathbf{c}_j \cdot \Delta_{1,j} \pmod{\varphi(N)}$. Note that $\Delta_{1,j}$ must be non-zero as $\boldsymbol{\gamma}_j^{(N-p-q)/4} \not\equiv \pm 1$

(mod N) but $\boldsymbol{\gamma}_j^{(N-p-q+\Delta_{1,j})/4} \equiv \pm 1 \pmod{N}$. Therefore, for any choice of $\Delta_{2,j}$ there is at most one $\mathbf{c}_j \in \{0,1\}$ that will satisfy $\Delta_{2,j} = \mathbf{c}_j \cdot \Delta_{1,j} \pmod{\varphi(N)}$. However, since $\mathbf{c}_j$ is sampled randomly in Step 5b of π_{biprime} *after* $\Delta_{2,j}$ is fixed, this equality is satisfied with probability at most $1/2$.

We can therefore bound the probability of the adversary succeeding in the j'th iteration as $\Pr[\text{bad } \boldsymbol{\gamma}_j] + \Pr[\text{good } \boldsymbol{\gamma}_j] \cdot \Pr[\mathcal{A} \text{ guesses } \mathbf{c}_j] = \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} = \frac{3}{4}$. The success probability in all $3s$ iterations is therefore at most $(3/4)^{3s} < 2^{-s}$. □

6 Efficiency Analysis

In this section we analyze the asymptotic cost of instantiating our template, and give concrete estimates for conservative parameters. We begin by analyzing the efficiency of CRTSample, in particular the probability of success of a single invocation, and the expected number of invocations to output a biprime. This directly tells us how to compose $\mathcal{F}_{\mathsf{RSAGen}}$; in sequence/parallel for a desired probability of success.

Following this we enumerate the cost of a single invocation of $\mathcal{F}_{\mathsf{RSAGen}}$. The template protocol π_{RSAGen} is divided into three logical phases: (1) candidate sieving, (2) biprimality testing, and (3) cheat detection. We count the number of calls made to the $\mathcal{F}_{\mathsf{AugMul}}$ and $\mathcal{F}_{\mathsf{Biprime}}$ oracles in each phase. At this point we are able to provide a direct comparison to [FLOP18]. We give the asymptotic and concrete costs of instantiating these instructions with both semi-honest and malicious security, and show comprehensively how we improve upon prior work.

6.1 Analysis of CRTSample

We state the relevant lemmas to upper bound the expected number of invocations of CRTSample below, and defer their proofs to Appendix E as they can be derived by standard number-theoretic techniques.

Lemma 6.1. CRTSample *succeeds with probability* $\Omega\left(\log^2 \kappa / \kappa^2\right)$.

Corollary 6.2. *The expected number of biprimes sampled after* $O(\kappa^2/\log^2 \kappa)$ *invocations of* CRTSample *is greater than one.*

Concretely, we analyze how these settings change for different moduli in Table 1.

6.2 Complexity in the $(\mathcal{F}_{\mathsf{AugMul}}, \mathcal{F}_{\mathsf{Biprime}})$-Hybrid Model

Our modular analysis allow us to express our protocol costs in terms of calls to $\mathcal{F}_{\mathsf{AugMul}}$ and $\mathcal{F}_{\mathsf{Biprime}}$. Note, these counts are the same in both the semi-honest and malicious models, and thus we use them below. As we envision future works relying on this template, but offering more efficient realizations of these two functionalities, we report the cost of each phase of our protocol template.

		$\kappa = 2048$	3072	4096
# Primes Sample (ℓ)	$O(\kappa/\log\kappa)$	131	183	233
# Primes Multiply ($\ell' - \ell$)	$O(\kappa/\log\kappa)$	103	145	186
Max Sample Prime ($\mathbf{m}_\ell$)	$\Omega(\kappa)$	743	1097	1481
Success Prob.	$\Omega(\frac{\log^2\kappa}{\kappa^2})$	$1/3630$	$1/7290$	$1/11910$

Table 1: Concrete CRT parameters for biprimes of size 2048, 3072, 4096.

1. **Sieving**: For each $j \in [2, \ell]$, the protocol template instructs parties to invoke the `sample` instruction of $\mathcal{F}_{\mathsf{AugMul}}$ using modulus $\mathbf{m}_j$. Following this, for $j \in [\ell+1, \ell']$ parties invoke the `input` instruction of $\mathcal{F}_{\mathsf{AugMul}}$ with modulus $\mathbf{m}_j$ twice, and then the `multiply` instruction with modulus $\mathbf{m}_j$.

2. **Biprimality Test**: Parties send `check-biprimality` to $\mathcal{F}_{\mathsf{Biprime}}$ at most once per instance of π_{RSAGen}.

3. **Consistency Check**: This phase has no cost in the semi-honest setting. In the malicious setting, depending on whether the parties are validating a candidate or verifying the legitimacy of a candidate, the parties either send `open` to $\mathcal{F}_{\mathsf{AugMul}}$ with modulus $\mathbf{m}_j$ for each $j \in [2, \ell']$, or they send `check`, f, respectively. The predicate f used in this phase comprises $2n$ modular additions over each $\mathbf{m}_i$ for $i \in \ell'$, two invocations of CRTRecon over the same set of moduli, one integer multiplication with inputs of length κ, and one equality check over values of length 2κ. The input size is $2 \cdot (n+1) \cdot 2\kappa$ bits, and the size in AND gates[5] of the predicate is

$$4\kappa^2 + 2\kappa + 2 \sum_{j\in[\ell']} (4 \cdot (n-1) \cdot |\mathbf{m}_j| + 8\kappa + 2 \cdot (|\mathbf{m}_j| + 1) \cdot \kappa)$$

6.3 Instantiating the Protocol Template

Given the number of calls to each method calculated above, we now discuss both the asymptotic and concrete cost estimates based protocols using state-of-the-art OT implementations for $\mathcal{F}_{\mathsf{AugMul}}$ (See Appendix C) as well as $\mathcal{F}_{\mathsf{Biprime}}$.

6.3.1 Semi-Honest

Our template only involves 4 cases; we summarize semi-honest costs in Table 2. Note, we omit `open` or `check` operations because they are not used. The concrete

[5]Taking modular addition over a modulus m to cost $4|m|$ gates, and taking asymmetric non-wrapping multiplication with one input less than m_1 and the other less than m_2 to cost $(|m_1| + 1) \cdot |m_2|)$ gates, and taking bit-shifts to be free. Along with equality test, these operations are sufficient to implement the predicate.

	Operation	Asymptotic	Concrete (KB)		
			2048	3072	4096
$\mathcal{F}_{\mathsf{AugMul}}$	`sample`	$O(\kappa \log \kappa)$	4	6	8
	`multiply`	$O(\kappa \log \kappa)$	4	7	9
$\mathcal{F}_{\mathsf{Biprime}}$	`not-biprime`	$O(\kappa)$	1	2	2
	`biprime`	$O(\kappa^2)$	1115	2435	4267

Table 2: Semi-honest communication costs for $\mathcal{F}_{\mathsf{AugMul}}$ and $\mathcal{F}_{\mathsf{Biprime}}$. These costs reflect per-party communication when multiplied by n.

values are computed by a python program written to simulate an execution and aggregate costs.

6.3.2 Malicious

Estimating the costs of malicious security requires further care to incorporate the additional steps required in each of the above four steps; in addition we need to estimate the costs of `open` and `check`. We briefly summarize the additional costs and then present them in Table 3. A more complete enumeration of $\mathcal{F}_{\mathsf{AugMul}}$ costs can be found in Appendix C.3

1. The maliciously secure `sample`, `multiply` methods in $\mathcal{F}_{\mathsf{AugMul}}$ involves a $2s$ bit encoding overhead for each multiplication, thus an ℓ-bit multiplication requires $\ell(\ell + 2s)$ bits of communication. This gives $O(\frac{\kappa}{\log \kappa} \cdot (s \log \kappa)) = O(\kappa s)$ overall complexity for these instructions.

2. Failed candidates require a call to `open` in $\mathcal{F}_{\mathsf{AugMul}}$ that requires sending $\ell(\ell + 2s)$ bits per input in the multiply. This gives $O(\frac{\kappa}{\log \kappa} \cdot (s \log \kappa)) = O(\kappa s)$ overall complexity for these instructions.[6]

3. Successful candidates require a call to `check` in $\mathcal{F}_{\mathsf{AugMul}}$ that requires using a generic MPC to evaluate a circuit of size $O(\kappa^2)$ gates, and performing an additional $s/\log \kappa$ calls to `multiply` for each prime for total cost $O(s^2\kappa/\log \kappa)$. We estimate the circuit size analytically in Appendix C.3. Cost estimates for the generic MPC assume that we make use of the generic MPC protocol of Hazay, Scholl, and Soria-Vazquez [HSS17]. Specifically, for a circuit of size C, with parameter $B = 1 + s/\log(C)$, with a per-OT cost of $O \approx .1$, we calculate the cost per gate as $(3B^2O + 4B^2 + B - 1 + 8 * \kappa_{GC}) \cdot \binom{n}{2}$.

4. Successful candidates must be subjected to the verification stage of the Biprimality test, and the cost of this stage is dominated by the cost of

[6]Meanwhile the equivalent multiplication in [FLOP18] is performed via running Gilboa's protocol composed of $2\kappa + 3s$ OT invocations with messages of size 2κ each, yielding a cost of $4\kappa^2 + 6\kappa s$ bits.

	Operation	Asymptotic	Concrete (MB)		
			2048	3072	4096
$\mathcal{F}_{\mathsf{AugMul}}$	`sample`	$O(s\kappa)$	.138	.204	.272
	`multiply`	$O(s\kappa)$	.682	.101	.136
	`open`	$O(s\kappa)$	.902	.134	.180
	`check`	$O(s\kappa^2)$ comm.	971	1686	2693
		$O(\kappa^2)$ gates	7304k	12.73m	20.40m
$\mathcal{F}_{\mathsf{Biprime}}$	`not-biprime`	$O(n\kappa)$	0	0	0
	`biprime`	$O(ns^2\kappa + s\kappa^2)$	770	1493	2768
		$O(ns\kappa + \kappa^2)$ gates	5770k	11.26m	20.97m

Table 3: Communication complexity for malicious secure instantiation of π_{RSAGen}. These costs reflect per-party communication when multiplied by n.

evaluating the VerifyBiprime predicate inside a generic MPC. The circuit representation of this predicate comprises

$$8n \cdot \kappa + 33\kappa^2 + 2n \cdot (\kappa - \log n + 1) + 6s \cdot n \cdot (2\kappa - 2\log n + s + 1)$$

AND gates,[7] 2κ output bits, and $3n \cdot s \cdot (\kappa - \log n + s + 1) + 2n \cdot (\kappa - \log n)$ input wires. Plugging these values into the cost equations for the generic MPC protocol Hazay, Scholl, and Soria-Vazquez yields a total asymptotic complexity of $O(n^3s^2\kappa + n^2s\kappa)$ for these instructions.

Total costs We have until this point analyzed the cost of instantiating $\mathcal{F}_{\mathsf{RSAGen}}$ for a single execution, however, one requires multiple invocations of $\mathcal{F}_{\mathsf{RSAGen}}$ *in parallel* to generate a biprime in expectation in constant rounds. The expected number of iterations is derived in Table 1. Note, there are natural strategies that trade-off number of rounds with communication overhead. We summarize the expected total communication costs to generate an RSA modulus in Table 4.

6.4 Comparison to Prior Work

In this section, we provide a comparison to the closest prior work [FLOP18] for the case of $n = 2$ parties. Communication costs are not listed in [FLOP18], so we estimate here based on the protocols as described using the state of the art OT constructions.

The main differences in efficiency arise because (a) we use an asymptotically faster multiplication and sieving functionality, and (b) our security functionality

[7] We take the cost of a modular addition over field $\mathbb{Z}_m$ to be $4|m|$ and the cost of a modular multiplication to be $\mathbb{Z}_m^2$

	Asymptotic	**Concrete (MB)**		
		2048	3072	4096
Semi-Honest	$O(\kappa^3/\log\kappa)$	32	100	225
Malicious	$O(s\kappa^3/\log^2\kappa)$	2812	6350	12130
	$O(ns\kappa + \kappa^2)$ gates	13m	24m	41m

Table 4: Total expected communication costs per-party to generate an RSA modulus using $\mathcal{F}_{\mathsf{RSAGen}}$. To estimate for n parties, multiply each number by n. Here we assume that $s > \log\kappa$ in the asymptotic analysis.

	Operation	**Asymptotic**	**Concrete (KB)**		
			2048	3072	4096
[FLOP18]	Div OT	$O(s\kappa/\log\kappa)$	5	8	10
	Multiply	$O(\kappa^2)$	512	1152	2049
This work	Sample	$O(\kappa\log\kappa)$	4	6	8
	Multiply	$O(\kappa\log\kappa)$	4	7	9

Table 5: Communication complexity in the semi-honest case for sieving.

lets us detect cheating that causes sampling failure whereas they require many extra repetitions to detect this event. These differences lead to substantial overhead in their protocol.

Semihonest protocol In the semi-honest case, we report the communication complexity to sieve an initial candidate. We use a similar $\mathcal{F}_{\mathsf{Biprime}}$ method and thus do not report its communication here. Our protocol's faster sieving process is roughly as follows:

In order to calculate the Div OT cost for [FLOP18] we assume that using 1-n random OT extension an 1-n random OT can be completed sending only one message. Additionally we calculate the expected number of iterations E using the equations given in Appendix B with an appropriate trial division bound. Since the Div OT protocol utilizes one OT and then sends 1 message, each of size κ_{OT} we calculate the total cost for Div OT as $2 * \kappa_{OT} * E$ where the trial division bounds match those used in our protocol.

Malicious protocol In comparison to [FLOP18], the differences become even more pronounced in the malicious setting. Their approach uses a GMW-style proof of honesty that we report in terms of boolean gates, using only the figure for $\kappa = 1024$ from their paper. However the major difference arises in the

			Cost (MB)	Reps	Total (GB)
[FLOP18]	Div OT	$O(s\kappa/\log\kappa)$	.005	204k	447
	Multiply	$O(\kappa^2)$	2.3		
	PoH	$O(s\kappa^2)$	904	1	
	PoH	$O(\kappa^2)$ gates	6.8m		
This work	Sample	$O(s\kappa)$	.138	3.6k	2.7
	Multiply	$O(s\kappa)$	.068		
	Check	$O(s\kappa^2)$	1730	1	
	Check	$O(\kappa^2)$ gates	13m		

Table 6: Communication complexity in the malicious model with $\kappa = 1024$.

number of repetitions required to sample a candidate. Whereas our protocol requires $O(\kappa/\log\kappa)$ iterations to achieve security with abort, we note here that [FLOP18] have no choice but to run $O(s(\kappa^2/\log^2\kappa))$ instances of candidate generation in the event that a malicious party executes a denial of service attack.

Bibliography

[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 *Advances in Cryptology – CRYPTO 2002*, pages 417–432, 2002.

[BCG+19] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure computation. In *Proceedings of the 26th ACM Conference on Computer and Communications Security, (CCS)*, pages 291–308, 2019.

[BF97] Dan Boneh and Matthew K. Franklin. Efficient generation of shared RSA keys. In *Advances in Cryptology – CRYPTO '97*, pages 425–439, 1997.

[BF01] Dan Boneh and Matthew K. Franklin. Efficient generation of shared RSA keys. *Journal of the ACM*, 48(4):702–722, 2001.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1–10, 1988.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 136–145, 2001.

[CIK$^+$20] Megan Chen, Yuval Ishai, Yuriy Kashnikov, Daniele Micciancio, Tarik Riviere, abhi shelat, Muthu Venkitasubramaniam, and Ruihan Wang. Diogenes: Lightweight scalable RSA modulus generation with a dishonest majority. Manuscript, 2020.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC)*, pages 494–503, 2002.

[Coc97] Clifford Cocks. Split knowledge generation of RSA parameters. In *Proceedings of the 6th International Conference on Cryptography and Coding*, pages 89–95, 1997.

[Coc98] Clifford Cocks. Split generation of RSA parameters with multiple participants. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.177.2600, 1998.

[DKLS18] Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. Secure two-party threshold ECDSA from ECDSA assumptions. In *Proceedings of the 39th IEEE Symposium on Security and Privacy, (S&P)*, pages 980–997, 2018.

[DKLS19] Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. Threshold ECDSA from ECDSA assumptions: The multiparty case. In *Proceedings of the 40th IEEE Symposium on Security and Privacy, (S&P)*, 2019.

[DM10] Ivan Damgård and Gert Læssøe Mikkelsen. Efficient, robust and constant-round distributed RSA key generation. In *Proceedings of the 7th Theory of Cryptography Conference, TCC 2010*, pages 183–200, 2010.

[FLOP18] Tore Kasper Frederiksen, Yehuda Lindell, Valery Osheter, and Benny Pinkas. Fast distributed RSA key generation for semi-honest and malicious adversaries. In *Advances in Cryptology – CRYPTO 2018, part II*, pages 331–361, 2018.

[FMY98] Yair Frankel, Philip D. MacKenzie, and Moti Yung. Robust efficient distributed RSA-key generation. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, page 320, 1998.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology – CRYPTO '86*, pages 186–194, 1986.

[Gav12] Gérald Gavin. RSA modulus generation in the two-party case. *IACR Cryptology ePrint Archive*, 2012:336, 2012.

[Gil99] Niv Gilboa. Two party RSA key generation. In *Advances in Cryptology – CRYPTO '99*, pages 116–129, 1999.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 218–229, 1987.

[Gol01] Oded Goldreich. *The Foundations of Cryptography - Volume 1: Basic Techniques.* Cambridge University Press, 2001.

[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, 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 *Topics in Cryptology - CT-RSA 2012 - The Cryptographers' Track at the RSA Conference*, pages 313–331, 2012.

[HSS17] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In *Advances in Cryptology – ASIACRYPT 2017, part I*, pages 598–628, 2017.

[IN96] Russell Impagliazzo and Moni Naor. Efficient cryptographic schemes provably as secure as subset sum. *Journal of Cryptology*, 9(4):199–216, 1996.

[JP99] Marc Joye and Richard Pinch. Cheating in split-knowledge rsa parameter generation. In *Workshop on Coding and Cryptography*, pages 157–163, 1999.

[KL15] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*, chapter Digital Signature Schemes, pages 443–486. Chapman & Hall/CRC, 2015.

[Knu69] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms.* Addison-Wesley, 1969.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In *Advances in Cryptology – CRYPTO 2015, part I*, pages 724–741, 2015.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 23th ACM Conference on Computer and Communications Security, (CCS)*, pages 830–842, 2016.

[Mil76] Gary L. Miller. Riemann's hypothesis and tests for primality. *J. Comput. Syst. Sci.*, 13(3):300–317, 1976.

[PS98] Guillaume Poupard and Jacques Stern. Generation of shared RSA keys by two parties. In *Advances in Cryptology – ASIACRYPT '98*, pages 11–24, 1998.

[Rab80] Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[Riv80] Ronald L. Rivest. A description of a single-chip implementation of the RSA cipher, 1980.

[Riv84] Ronald L. Rivest. RSA chips (past/present/future). In *Workshop on the Theory and Application of Cryptographic Techniques*, pages 159–165. Springer, 1984.

[RS62] J. Barkley Rosser and Lowell Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois J. Math.*, 6:64–94, 1962.

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

A BFGM

Algorithm A.1. BFGM(n, M)

1. For $i \in [n]$, sample $p_i \leftarrow [0, M)$ and $q_i \leftarrow [0, M)$ subject to $p_1 \equiv q_1 \equiv 3 \pmod 4$ and $p_j \equiv q_j \equiv 0 \pmod 4$ for $j \in [2, n]$.
2. Compute
$$p := \sum_{i\in[n]} p_i \qquad \text{and} \qquad q := \sum_{i\in[n]} q_i \qquad \text{and} \qquad N := p \cdot q$$
3. If $\gcd(N, p + q - 1) = 1$, and both p and q are primes, then output $(N, \{(p_i, q_i)\}_{i\in[n]})$. Otherwise, repeat this procedure from Step 1 until these conditions are satisfied.

B The UC Model and Useful Functionalities

B.1 Universal Composability

We give a high-level overview of the UC model and refer the reader to [Can01] for a further details.

The *real-world* experiment involves n parties $\mathcal{P}_1, \dots, \mathcal{P}_n$ that execute a protocol π, an adversary $\mathcal{A}$ that can corrupt a subset of the parties, and an environment $\mathcal{Z}$ that is initialized with an advice-string z. All entities are initialized with the security parameter κ and with a random tape. The environment activates the parties involved in π, chooses their inputs and receives their outputs, and communicates with the adversary $\mathcal{A}$. A *semi-honest* adversary simply observes the memory of the corrupted parties, while a *malicious* adversary may instruct them to arbitrarily deviate from π. In this work, we consider only *static* adversaries, who corrupt up to $n-1$ parties at the beginning of the experiment. The real-world experiment completes when $\mathcal{Z}$ stops activating other parties and outputs a decision bit. Let $\textsc{real}_{\pi,\mathcal{A},\mathcal{Z}}(z, \kappa)$ denote the random variable representing the output of the experiment.

The *ideal-world* experiment involves n dummy parties $\mathcal{P}_1, \dots, \mathcal{P}_n$, an ideal functionality $\mathcal{F}$, an ideal-world adversary $\mathcal{S}$ (the simulator), and an environment $\mathcal{Z}$. The dummy parties act as routers that forward any message received from $\mathcal{Z}$ to $\mathcal{F}$ and vice versa. The simulator can corrupt a subset of the dummy parties and interact with $\mathcal{F}$ on their behalf; in addition, $\mathcal{S}$ can communicate directly with $\mathcal{F}$ according to its specification. The environment and the simulator can interact throughout the experiment, and the goal of the simulator is to trick the environment to believe it is running in the real experiment. The ideal-world experiment completes when $\mathcal{Z}$ stops activating other parties and outputs a decision bit. Let $\textsc{ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(z, \kappa)$ denote the random variable representing the output of the experiment.

A protocol π UC-realizes a functionality $\mathcal{F}$ if for every probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ there exists a PPT simulator $\mathcal{S}$ such that for every PPT environment $\mathcal{Z}$

$$\left\{\text{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(z,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \approx_{\mathrm{c}} \left\{\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(z,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}}$$

Communication model. We follow standard practice of MPC protocols: Every pair of parties can communicate via an authenticated channel, and in the malicious setting we additionally assume the existence of a broadcast channel. Formally, the protocols are defined in the $(\mathcal{F}_{\mathsf{auth}}, \mathcal{F}_{\mathsf{bc}})$-hybrid model (see [Can01, CLOS02]). We leave this implicit in their descriptions.

B.2 Useful Functionalities

To realize $\mathcal{F}_{\mathsf{AugMul}}$ and $\mathcal{F}_{\mathsf{Biprime}}$, we use a number of standard or nearly-standard functionalities that have well-known realizations in the cryptography literature. For completeness, we give those functionalities in this section. First among them is a simple distributed coin-tossing functionality, which samples from an arbitrary domain.

Functionality B.1. $\mathcal{F}_{\mathsf{CT}}(n, \mathbb{X})$. **Coin Tossing**

This functionality is parameterized by the number of parties n and a domain $\mathbb{X}$.

Sample: Upon receiving $(\texttt{flip}, \mathsf{sid})$ from all parties, where sid is a fresh, agreed-upon value, uniformly sample a random element $x \leftarrow \mathbb{X}$ and send $(\texttt{coin}, \mathsf{sid}, x)$ to all parties as adversarially delayed output.

We also make use of a one-to-many commitment functionality, which we have taken directly from [CLOS02].

Functionality B.2. $\mathcal{F}_{\mathsf{Com}}(n)$. **One-to-many Commitment**

This functionality is parameterized by the number of parties n. In each instance one specific party $\mathcal{P}_i$ commits, and all other parties receive the commitment and committed value.

Commit: On receiving $(\texttt{commit}, \mathsf{sid}, x, \mathbf{D})$ from party $\mathcal{P}_i$, where $\mathbf{D} \subseteq [n]$ and $x \in \{0,1\}^*$, if sid is a fresh value, then store $(\texttt{commitment}, \mathsf{sid}, x, \mathbf{D}, i)$ in memory and send $(\texttt{committed}, \mathsf{sid}, i)$ to each party $\mathcal{P}_j$ for $j \in \mathbf{D}$.

Decommit: On receiving $(\texttt{decommit}, \mathsf{sid})$ from $\mathcal{P}_i$, if a record of the form $(\texttt{commitment}, \mathsf{sid}, x, \mathbf{D}, i)$ exists in memory, then send $(\texttt{decommitted}, \mathsf{sid}, x)$ to every party $\mathcal{P}_j$ for $j \in \mathbf{D}$.

In π_{biprime}, we make use of a functionality for randomly sampling integer shares of zero. This functionality can be realized generically, and Algesheimer et al. [ACS02] give a bespoke protocol that we conjecture also realizes it (though we omit a proof).

Functionality B.3. $\mathcal{F}_{\mathsf{Zero}}(n, B)$. **Integer Zero Sharing**

This functionality is parameterized by a number of parties n and a maximal value B.

Sample: Upon receiving $(\texttt{sample}, \mathsf{sid})$ from all parties, where sid is a fresh, agreed-upon value, uniformly sample $x_i \leftarrow [-B, B]$ for each $i \in [n]$ conditioned on

$$\sum_{i \in [n]} x_i = 0$$

where the sum is computed over the integers, and send $(\texttt{zero-share}, \mathsf{sid}, x_i)$ to each party $\mathcal{P}_i$ as adversarially delayed private output.

In both π_{biprime} and π_{AugMul}, we use a functionality for generic commit-and-compute multiparty computation. This functionality allows each party to commit to private inputs, after which the parties agree on one or more arbitrary circuits to apply to those inputs. It can be realized using many generic multiparty computation protocols.

Functionality B.4. $\mathcal{F}_{\mathsf{ComCompute}}(n)$. **Commit-and-compute MPC**

This functionality is parameterized by the number of parties n.

Input Commitment: Upon receiving $(\texttt{commit}, \mathsf{sid}, x)$ from party $\mathcal{P}_i$, if sid is a fresh value, then store $(\texttt{value}, \mathsf{sid}, i, x)$ in memory, and send $(\texttt{committed}, \mathsf{sid}, i)$ to all other parties.

Computation: Upon receiving $(\texttt{compute}, \mathsf{sid}, \textsf{input-sids}, f)$ from all parties, where sid is a fresh, agreed upon value, and where $\textsf{input-sids}$ is a vector of session IDs such that for every $i \in [|\textsf{input-sids}|]$ there exists in memory a record of the form $(\texttt{value}, \textsf{input-sids}_i, *, *)$, and where f is the description of a function that takes as input the values associated with the IDs in $\textsf{input-sids}$ and produces as output an n-tuple of values, if the parties disagree upon the function f or the vector $\textsf{input-sids}$, then abort, informing them in an adversarially delayed fashion, and otherwise:

1. Let $\mathbf{x}$ be a vector of the same length as $\textsf{input-sids}$ such that for $i \in [|\textsf{input-sids}|]$, there exists in memory a record of the form $(\texttt{value}, \textsf{input-sids}_i, *, v)$ such that $\mathbf{x}_i = v$.
2. Compute $(y_1, \ldots, y_n) := f(\mathbf{x})$, and then send $(\texttt{result}, \mathsf{sid}, y_i)$ to each party $\mathcal{P}_i$ as an adversarially delayed private output.

Finally, we use a Delayed-transmission Correlated Oblivious Transfer functionality as the basis of our multiplication protocols. This functionality can be realized by combining a standard COT protocol with a commitment scheme. Unlike an ordinary COT functionality, which allows the sender to associate a single correlation to each choice bit, this functionality allows the sender to associate an arbitrary number of correlations to each bit. The transfer action then commits the sender to the correlations, and the sender can later decommit them individually, at which point the receiver learns either a random pad, or the same pad plus the decommitted correlation, according to their choice. We suggest that this functionality be instantiated via either Silent OT [BCG+19] or the KOS OT-extension protocol [KOS15]. A formal description of such an instantiation will be provided in the full version of this document.

Functionality B.5. $\mathcal{F}_{\mathsf{DelayedCOT}}$**. Delayed-transmission COT**

This functionality interacts with a sender S and a receiver R.

Receiver Choice: On receiving $(\texttt{choose}, \mathsf{sid}, \beta)$ from the receiver, where $\beta \in \{0,1\}$, if sid is a fresh value, then store $(\texttt{choice}, \mathsf{sid}, \beta)$ in memory and send $(\texttt{chosen}, \mathsf{sid})$ to the sender.

Sender Commitment: On receiving $(\texttt{commit}, \mathsf{sid}, \boldsymbol{\alpha})$ from the sender, where $\boldsymbol{\alpha} \in \mathbb{G}_1 \times \mathbb{G}_2 \times \ldots$ (that is, $\boldsymbol{\alpha}$ is a vector of elements from a heterogeneous set of groups), if a record of the form $(\texttt{choice}, \mathsf{sid}, *)$ exists in memory, but no record of the form $(\texttt{correlation}, \mathsf{sid}, *)$ exists in memory, then store $(\texttt{correlation}, \mathsf{sid}, \boldsymbol{\alpha})$ in memory and send $(\texttt{committed}, \mathsf{sid})$ to the receiver.

Transfer: On receiving $(\texttt{transfer}, \mathsf{sid}, i)$ from the sender, if records of the form $(\texttt{correlation}, \mathsf{sid}, \boldsymbol{\alpha})$ and $(\texttt{choice}, \mathsf{sid}, \beta)$ exist in memory such that $0 < i \leq |\boldsymbol{\alpha}|$, but no record $(\texttt{complete}, \mathsf{sid}, i)$ exists in memory, then, taking $\mathbb{G}$ to be the group in which $\boldsymbol{\alpha}_i$ resides, sample a random pad $\rho \leftarrow \mathbb{G}$, send $(\texttt{pad}, \mathsf{sid}, i, \rho)$ to the sender and $(\texttt{message}, \mathsf{sid}, i, \rho + \beta \cdot \boldsymbol{\alpha}_i)$ to the receiver, where the $+$ operator is defined over $\mathbb{G}$, and store $(\texttt{complete}, \mathsf{sid}, i)$ in memory.

C Instantiating Multiplication

In this section, we describe how to instantiate the $\mathcal{F}_{\mathsf{AugMul}}$ functionality, and discuss the efficiency of our protocols. We begin by using oblivious transfer ($\mathcal{F}_{\mathsf{DelayedCOT}}$) to realize a two-party multiplier $\mathcal{F}_{\mathsf{ReuseMul2P}}$ that allows inputs to be reused, and postpones the detection of malicious behaviour. We then plug this into the classic GMW [GMW87] multiplication technique in order to realize an n-party multiplier $\mathcal{F}_{\mathsf{ReuseMul}}$, with the same properties of input-reuse and delayed cheat detection. Finally, we combine this component with generic multiparty computation ($\mathcal{F}_{\mathsf{ComCompute}}$) via a simple MAC to realize $\mathcal{F}_{\mathsf{AugMul}}$. We begin describing these components from $\mathcal{F}_{\mathsf{ReuseMul2P}}$ onward.

C.1 Two-party Reusable-input Multiplier

Our basic two-party multiplication functionality $\mathcal{F}_{\mathsf{ReuseMul2P}}$ allows parties to input arbitrarily many values, whereafter, on request, it returns additive shares of the product of any pair of them. Unlike the standard two-party multiplication functionality, however, we allow the adversary to both request the honest party's inputs and determine the output products, and then add an explicit check command which notifies the honest party of such behaviour if called. In addition, we give the functionality an interface by which the parties can agree to open their private inputs to each other (which, as a side effect, also notifies the honest party of any cheating behavior).

Functionality C.1. $\mathcal{F}_{\mathsf{ReuseMul2P}}(m)$**. Two-party Multiplication**

This functionality is parameterized by the prime modulus m. It interacts with two parties, Alice and Bob, who have indices A and B respectively, and it also interacts directly with an ideal adversary $\mathcal{S}$, who corrupts one of the parties. The index of the honest party is given by h, and the index of the corrupt party is given by c.

Cheater Activation: Upon receiving $(\texttt{cheat}, \mathsf{sid})$ from $\mathcal{S}$, store $(\texttt{cheater}, \mathsf{sid})$ in memory and send every record of the form $(\texttt{value}, \mathsf{sid}, i, x)$ to $\mathcal{S}$. For the purposes of this functionality, we will consider session IDs to be fresh even when a `cheater` record already exists in memory.

Input: Upon receiving $(\texttt{input-self}, \mathsf{sid}, x)$ from party $\mathcal{P}_i$, where $i \in \{\mathsf{A}, \mathsf{B}\}$, and also receiving $(\texttt{input-other}, \mathsf{sid})$ from the opposite party: if sid is a fresh, agreed-upon value and if $0 \leq x < m$, then store $(\texttt{value}, \mathsf{sid}, i, x)$ in memory and send $(\texttt{value-loaded}, \mathsf{sid})$ to both parties. If a record of the form $(\texttt{cheater}, \mathsf{sid})$ exists in memory, then send $(\texttt{value}, \mathsf{sid}, i, x)$ to $\mathcal{S}$.

Multiplication: Upon receiving $(\texttt{multiply}, \mathsf{sid}_1, \mathsf{sid}_2, \mathsf{sid}_3)$ from $\mathcal{P}_h$ and $(\texttt{adv-multiply}, \mathsf{sid}_1, \mathsf{sid}_2, \mathsf{sid}_3, z_c)$ from $\mathcal{S}$ where $0 \leq z_c < m$,[a] if all three session IDs are agreed upon and sid_3 is fresh, and if records of the form $(\texttt{value}, \mathsf{sid}_1, i, x)$ and $(\texttt{value}, \mathsf{sid}_2, j, y)$ exist in memory such that $i \neq j$, and if no record of the form $(\texttt{cheater}, \mathsf{sid}_1)$ or $(\texttt{cheater}, \mathsf{sid}_2)$ exists in memory, then let $z_h := (x \cdot y - z_c) \bmod m$. If the previous conditions hold, but a record of the form $(\texttt{cheater}, \mathsf{sid}_1)$ or $(\texttt{cheater}, \mathsf{sid}_2)$ exists in memory, then send $(\texttt{cheat-multiply}, \mathsf{sid}_1, \mathsf{sid}_2, \mathsf{sid}_3)$ to $\mathcal{S}$ and in response receive $(\texttt{cheat-product}, \mathsf{sid}_3, z_h)$ where $0 \leq z_h < m$. Regardless, send $(\texttt{product}, \mathsf{sid}_3, z_{\mathsf{A}})$ and $(\texttt{product}, \mathsf{sid}_3, z_{\mathsf{B}})$ to Alice and Bob respectively as adversarially delayed private output.

Cheater Check: Upon receiving $(\texttt{check}, \mathsf{sid})$ from both parties, if a record of the form $(\texttt{cheater}, \mathsf{sid})$ exists in memory, then abort, informing both parties in an adversarially-delayed fashion. Otherwise, send $(\texttt{no-cheater}, \mathsf{sid})$ to both parties as adversarially-delayed private output. Regardless, refuse all future messages with this sid.

Input Revelation: Upon receiving $(\texttt{open}, \mathsf{sid})$ from both parties, if a record of the form $(\texttt{cheater}, \mathsf{sid})$ exists in memory, then abort, informing both parties in an adversarially delayed fashion. Otherwise, if a record of the form $(\texttt{value}, \mathsf{sid}, i, x)$ exists in memory, then send $(\texttt{opening}, \mathsf{sid}, x)$ to $\mathcal{P}_j$, where $j \in \{\mathsf{A}, \mathsf{B}\}$ and $j \neq i$, as adversarially delayed output. Refuse all future messages with this sid.

[a] In the semi-honest setting, the adversary does not send z_c to the functionality; instead the functionality samples the share for the corrupt party just as it does for honest party.

To realize $\mathcal{F}_{\mathsf{ReuseMul2P}}$ with a protocol $\pi_{\mathsf{mult\text{-}2P}}$, we adapt the multiplication protocols of Doerner et al. [DKLS19]. Here we give an informal description of the implementation of each phase, along with a security argument. The full version of this paper will have a formal description, and in the meantime we refer the reader to Doerner et al. [DKLS19, Section 3] for a more in-depth technical explanation.

Common Parameters and Hybrid Functionalities. $\pi_{\mathsf{mult\text{-}2P}}$is parameterized by the statistical security parameter s and a prime $m \in O(\log s)$. For convenience, we define a batch-size $\eta := 2s + |m|$, a repetition count $r = \lceil s/|m| \rceil$, and a randomly-sampled public gadget vector $\mathbf{g} \leftarrow \mathbb{Z}_m^{\eta}$. The participating parties have access to the commitment functionality $\mathcal{F}_{\mathsf{Com}}$ and the delayed-transfer OT functionality $\mathcal{F}_{\mathsf{DelayedCOT}}$. We suggest that $\mathcal{F}_{\mathsf{DelayedCOT}}$ be instantiated via either Silent OT [BCG+19] or the KOS OT-extension protocol [KOS15]. In the latter case, efficiency will be impacted by an additional symmetric computational security parameter, κ_{OT}. In practice, it is reasonable to assume that $\kappa_{\mathsf{OT}} \lesssim 2s$.

Inputs and Multiplication. For the sake of succinctness, we will describe the input and multiplication processes jointly: each party will supply exactly one input, and they will receive shares of the product, in a single step. Later, we will discuss how the parties can input values independently, and how those input values can be reused. Alice begins the protocol with an input $a \in [0, m)$, and Bob with an input $b \in [0, m)$, and they both know a fresh, agreed-upon Session ID sid. They take the following steps:

1. Alice and Bob both independently compute a vector of Session IDs $\textsf{bit-sids} := \{\mathsf{GenSID}(\mathsf{sid}, j)\}_{j \in [\eta]}$.
2. Alice samples a consistency check vector $\tilde{\mathbf{a}} \leftarrow \mathbb{Z}_m^r$.
3. Bob samples a vector of choice bits $\boldsymbol{\beta} \leftarrow \{0,1\}^{\eta}$.
4. For each $i \in [\eta]$ (in parallel):
   (a) Bob sends $(\texttt{choose}, \textsf{bit-sids}_i, \boldsymbol{\beta}_i)$ to $\mathcal{F}_{\mathsf{DelayedCOT}}$, and Alice is notified.

(b) Alice sends $(\texttt{commit}, \textsf{bit-sids}_i, \{a\} \| \tilde{\mathbf{a}})$ to $\mathcal{F}_{\mathsf{DelayedCOT}}$, and Bob is notified.

(c) Alice sends $(\texttt{transfer}, \textsf{bit-sids}_i, 1)$ to $\mathcal{F}_{\mathsf{DelayedCOT}}$. As a consequence, she receives $\mathbf{z}_{\mathsf{A},i}$ from $\mathcal{F}_{\mathsf{DelayedCOT}}$, and Bob receives $\mathbf{z}_{\mathsf{B},i}$, such that $\mathbf{z}_{\mathsf{A},i} + \mathbf{z}_{\mathsf{B},i} \equiv a \cdot \boldsymbol{\beta}_i \pmod{m}$.

5. Alice uses $\mathcal{F}_{\mathsf{Com}}$ to commit to $(a, \mathbf{z}_{\mathsf{A},*})$, and Bob is notified.

6. Bob computes $\tilde{b} \coloneqq \langle \mathbf{g}, \boldsymbol{\beta} \rangle$ and sends $\delta \coloneqq b - \tilde{b} \bmod m$ to Alice.

7. Alice outputs $z_{\mathsf{A}} \coloneqq a \cdot \delta + \langle \mathbf{g}, \mathbf{z}_{\mathsf{A},*} \rangle \bmod m$, while Bob outputs $z_{\mathsf{B}} \coloneqq \langle \mathbf{g}, \mathbf{z}_{\mathsf{B},*} \rangle \bmod m$

While the correctness of the above procedure is easy to verify when both parties follow the protocol, we note that it omits the consistency check components of the protocols of Doerner et al., which will appear in the next protocol phase. In particular, the consistency check vector $\tilde{\mathbf{a}}$ is committed by the end of the protocol, but it has not yet been transferred to Bob. This omission admits attacks, such as a corrupt Alice using different values for a in each iteration of Step 4b. We model these attacks in $\mathcal{F}_{\mathsf{ReuseMul2P}}$ by allowing the ideal adversary to fully control the results of a multiplication, once it has explicitly notified $\mathcal{F}_{\mathsf{ReuseMul2P}}$ that it wishes to cheat.

If the parties agree that Alice should be compelled to reuse an input in multiple different multiplications, then she must also reuse the same consistency check vector $\tilde{\mathbf{a}}$ in all of those multiplications. The consistency check mechanism (in the next protocol phase) that ensures the internal consistency of a single multiplication will also sure the consistency of many multiplications.

If the parties agree that Bob should be compelled to reuse an input in multiple different multiplications, then the above protocol is run exactly once, and Alice combines her inputs for all of those multiplications (and their associated, independent consistency check vectors) into a single array, which she commits to in Step 4b. She then repeats Step 4c, changing the index as appropriate to cause the transfer of each of her inputs (but not, for now, the consistency check vectors). The remaining steps in the protocol are repeated once for each multiplication, except for Step 6, which is performed exactly once, for Bob's one input.

In the case that only Bob inputs a value, the parties can run the above protocol until Step 4a is complete, and then pause the protocol until a multiplication using Bob's input must be performed. In the case that only Alice inputs a value, Bob must input a dummy value, and compulsory input reuse is employed to ensure she uses her input in the appropriate multiplications.

We will briefly review the relevant parts of the simulation argument of Doerner et al. [DKLS19]. Simulation against a corrupt Bob is simple: he has no avenue for cheating, and his ideal input in any single instance of the above protocol is defined by $b = \delta + \langle \mathbf{g}, \boldsymbol{\beta} \rangle \bmod m$. Simulating against a corrupt Alice is more involved: if she uses inconsistent values of a in Step 4b, then the simulator takes her most common value to be her ideal input, and tosses a coin for each inconsistency. If any coin returns 1, then the simulator activates the `cheat` interface of $\mathcal{F}_{\mathsf{ReuseMul2P}}$, receives Bob's inputs, and can thereafter behave exactly as he would in the real world. If all coins return 0, then the simulator samples a uniform value for $\delta \leftarrow \mathbb{Z}_m$ and sends it to Alice. Doerner et al. [DKLS19] show via a lemma of Impagliazzo and Naor [IN96] that this simulated δ is distinguishable from the real one with probability no greater than 2^{-s}.

Finally, we observe that the dominant cost of the above protocol is incurred by the $\eta = 2s+|m|$ invocations of $\mathcal{F}_{\mathsf{DelayedCOT}}$ per multiplication, each invocation with a correlations of size $|m|$. If we realize $\mathcal{F}_{\mathsf{DelayedCOT}}$ via the Silent OT protocol of Boyle et al. [BCG+19], then Alice must transmit $|m|(|m|+2s)$ bits, Bob must transmit $|m|$ bits, and they require two messages in total. If we realize $\mathcal{F}_{\mathsf{DelayedCOT}}$ via the KOS OT-extension protocol [KOS15], then Alice must transmit $|m|(|m|+2s)$ bits, Bob must transmit $\kappa_{\mathsf{OT}} \cdot (|m|+2s) \lesssim 2s \cdot (|m|+2s)$ bits, and they require three messages in total.

Cheater Check. In this phase of the protocol, the parties perform a process analogous to the consistency check in the multiplication protocols of Doerner et al. [DKLS19]. This reveals to the honest parties any cheats in the protocol phases described above, and corresponds to the Cheater Check phase in $\mathcal{F}_{\mathsf{AugMul}}$. As we have previously noted, Bob does not have an opportunity to cheat; thus this check leverages the consistency check vector $\tilde{\mathbf{a}}$, to which Alice is committed, in order to verify her behavior. In addition to the consistency check vector, Alice begins the protocol with an input a, and both parties know a vector sids of all the multiplications in which Alice was expected to use this input, along with a vector $\mathsf{bit\text{-}sids}$ of the individual $\mathcal{F}_{\mathsf{DelayedCOT}}$ instances (over all of the multiplications associated with sids) in which she was expected to commit $a\|\tilde{\mathbf{a}}$. Bob begins with a vector of choice bits, $\boldsymbol{\beta}$, one bit for each entry in $\mathsf{bit\text{-}sids}$. We assume for the sake of simplicity that Bob was not expected to reuse his inputs. The parties take the following steps:

1. For each $i \in [|\mathsf{bit\text{-}sids}|]$ and $j \in [r]$, Alice sends $(\mathtt{transfer}, \mathsf{bit\text{-}sids}_i, j+1)$ to $\mathcal{F}_{\mathsf{DelayedCOT}}$ and receives $\tilde{\mathbf{z}}_{\mathsf{A},i,j}$ in response such that $0 \le \tilde{\mathbf{z}}_{\mathsf{A},i,j} < m$, while Bob receives $\tilde{\mathbf{z}}_{\mathsf{B},i,j}$ such that $0 \le \tilde{\mathbf{z}}_{\mathsf{B},i,j} < m$. Note that if Alice has behaved honestly, then per the specification of $\mathcal{F}_{\mathsf{DelayedCOT}}$, it holds for all $i \in [|\mathsf{bit\text{-}sids}|]$ and $j \in [r]$ that
$$\tilde{\mathbf{z}}_{\mathsf{A},i,j} + \tilde{\mathbf{z}}_{\mathsf{B},i,j} \equiv \boldsymbol{\beta}_i \cdot \tilde{\mathbf{a}}_j \pmod{m}$$
2. Bob sends a random challenge $\mathbf{e} \leftarrow \mathbb{Z}_m^r$ to Alice.
3. Alice computes her responses
$$\boldsymbol{\psi} := \{(\tilde{\mathbf{a}}_j + \mathbf{e}_j \cdot a) \bmod m\}_{j\in[r]}$$
$$\boldsymbol{\zeta} := \{(\tilde{\mathbf{z}}_{\mathsf{A},i,j} + \mathbf{e}_j \cdot \mathbf{z}_{\mathsf{A},i}) \bmod m\}_{i\in[|\mathsf{bit\text{-}sids}|],\, j\in[r]}$$
4. Bob verifies that for each $i \in [|\mathsf{bit\text{-}sids}|]$ and $j \in [r]$,
$$\tilde{\mathbf{z}}_{\mathsf{B},i,j} + \mathbf{e}_j \cdot \mathbf{z}_{\mathsf{B},i} \equiv \boldsymbol{\zeta}_{i,j} - \boldsymbol{\beta}_i \cdot \boldsymbol{\psi}_j \pmod{m}$$

Notice that in the foregoing procedure, Bob learns nothing about α or $\mathbf{z}_{\mathsf{A}}$; all values derived from these are masked by Alice using uniformly-sampled one-time pads before being transmitted to him. Notice also that the equality in Step 4 will fail for some index $i \in [|\textsf{bit-sids}|]$ if and only if Alice has used a value other than $\alpha \| \mathbf{z}_{\mathsf{A}}$ as her correlation in that instance of $\mathcal{F}_{\mathsf{DelayedCOT}}$, *and* $\boldsymbol{\beta}_i = 1$. Thus, if she guesses Bob's choice bits correctly and cheats only where he has chosen 0, her cheats will go undetected (and since the bits are 0, her cheats will have no effect on the output). Alice can leverage this fact to learn some of Bob's choice bits via selective failure. Fortunately, Bob's choice bits are chosen by encoding his true input, using a random public vector $\mathbf{g}$. Doerner et al. [DKLS19] prove that under this encoding scheme, she has a negligible chance to learn enough bits to distinguish his input from a uniform string with noticeable probability. We refer the reader to their work for a full explanation.

This protocol can be optimized (at the cost of straight-line extractability and therefore UC-security) by applying the Fiat-Shamir transform [FS86] to generate the challenge $\mathbf{e}$. When either KOS OT-extension [KOS15] or Silent OT [BCG+19] is used to realize $\mathcal{F}_{\mathsf{DelayedCOT}}$, and this optimization is applied, the cheater check costs only one round and $2s \cdot (|m| + 2s) \cdot c$ bits of transmitted data for Alice, where c is the number of multiplications in which the input to be checked was used.

Input Revelation. In this phase of the protocol, the parties can open their inputs to one another. This phase may be run in place of the cheater check phase, but they may not both be run. It has no analogue in the protocols of Doerner et al. [DKLS18, DKLS19]. For the sake of simplicity, we assume that both parties wish to reveal their inputs simultaneously, though the protocol may be extended to allow independent release. Alice begins the protocol with an input a and a consistency check vector $\tilde{\mathbf{a}}$, to which Alice is committed, and an output vector $\mathbf{z}_{\mathsf{A}}$. Bob begins with a vector of choice bits, $\boldsymbol{\beta}$ and an output vector $\mathbf{z}_{\mathsf{B}}$. In addition, they both know the vector **bit-sids** of Session IDs of all relevant $\mathcal{F}_{\mathsf{DelayedCOT}}$ instances (i.e., one instance for each of Bob's bits, where Alice was expected to use the same input in all instances). The parties take the following steps:

1. For each $i \in [|\textsf{bit-sids}|]$ and $j \in [r]$, Alice sends $(\texttt{transfer}, \textsf{bit-sids}_i, j+1)$ to $\mathcal{F}_{\mathsf{DelayedCOT}}$ and receives $\tilde{\mathbf{z}}_{\mathsf{A},i,j}$ in response such that $0 \leq \tilde{\mathbf{z}}_{\mathsf{A},i,j} < m$, while Bob receives $\tilde{\mathbf{z}}_{\mathsf{B},i,j}$ such that $0 \leq \tilde{\mathbf{z}}_{\mathsf{B},i,j} < m$.

2. In order to prove that he used an input b, for each $i \in [|\textsf{bit-sids}|]$, Bob sends $(\boldsymbol{\beta}_i, \tilde{\mathbf{z}}_{\mathsf{B},i,*})$ to Alice, who verifies for each $i \in [|\textsf{bit-sids}|]$ and $j \in [r]$ that
$$\tilde{\mathbf{z}}_{\mathsf{A},i,j} + \tilde{\mathbf{z}}_{\mathsf{B},i,j} \equiv \boldsymbol{\beta}_i \cdot \tilde{\mathbf{a}}_j \pmod{m}$$
If these equations hold, then Alice also verifies that
$$\delta \equiv \langle \mathbf{g}, \beta \rangle - b \pmod{m}$$

The security of this step lies in the inherent committing nature of OT; Bob is able to pass the test while also lying about his choice bit (without loss of generality, $\boldsymbol{\beta}_i$) only by outright guessing the value for $\tilde{\mathbf{z}}_{\mathsf{B},i,*}$ that will cause the test to pass. This is as hard as guessing $\tilde{\mathbf{a}}$, and Bob succeeds with probability less than 2^{-s}.

3. In order to prove that she used an input a for all $\mathcal{F}_{\mathsf{DelayedCOT}}$ instances associated with the Session IDs in $\mathsf{bit\text{-}sids}$, Alice decommits $(a, \{\mathbf{z}_{\mathsf{A},i}\}_{i\in\mathsf{bit\text{-}sids}})$ via $\mathcal{F}_{\mathsf{Com}}$. Bob verifies that for each $i \in [|\mathsf{bit\text{-}sids}|]$, it holds that

$$\mathbf{z}_{\mathsf{A},i} + \mathbf{z}_{\mathsf{B},i} \equiv a \cdot \boldsymbol{\beta}_i \pmod{m}$$

Alice is able to subvert this check for some index i if and only if she correctly guesses Bob's corresponding choice bit $\boldsymbol{\beta}_i$ during the input phase, and appropriately offsets $\mathbf{z}_{\mathsf{A},i}$. We again refer the reader to Doerner et al. [DKLS19] for a full explanation.

We note that the foregoing protocol can only be simulated for small fields. Specifically, we require that $|m| \in O(\log s)$. This is due to the fact that opening Bob's input to some value b requires the simulator to compute $\hat{b} = \delta + b$ and then find $\boldsymbol{\beta} \in \{0,1\}^\eta$ such that $\langle \mathbf{g}, \boldsymbol{\beta} \rangle \equiv \hat{b} \pmod{m}$ (and such that any choice bit already fixed by a coin that the simulator flipped in the input phase is consistent). The brute force approach to finding such a vector of choice bits (i.e. guess and check) succeeds in polynomial time with overwhelming probability only when each individual guess satisfies the predicate with probability $\Omega(1/\operatorname{poly}(s))$. Given a randomly chosen $\mathbf{g} \in \mathbb{Z}_m^{|m|+2s}$, it follows from Impagliazzo and Naor [IN96] that a random sampling of $\boldsymbol{\beta} \leftarrow \{0,1\}^{|m|+2s}$ satisfies the predicate with probability no less than $1/m - 2^{-s} \in \Omega(1/\operatorname{poly}(s))$. As we have mentioned, Alice's (undetected) cheats in the input phase trigger coin flips that fix some of the simulator's choice bits. In the full version of this document, we will show that Alice has a negligible chance to prevent the simulator from finding a satisfying assignment in the ideal world while also avoiding an abort in the real world.

In the Random Oracle Model, this protocol can be optimized by having Bob send a $2s$-bit digest of his $\tilde{\mathbf{z}}_{\mathsf{B},*,*}$ values, instead of sending the values themselves, since Alice is able to recompute the same values herself using only $\boldsymbol{\beta}$ and the information already in her view. Under this optimization, the cost of this protocol is $|m| + 2s$ bits for Bob, $|m|(|m| + 2s)$ bits for Alice, and 3 messages in total.

C.2 n-party Reusable-input Multiplier

Plugging $\mathcal{F}_{\mathsf{ReuseMul2P}}$ into a GMW-style [GMW87] multiplication protocol yields an n-party equivalent of the same functionality, i.e. $\mathcal{F}_{\mathsf{ReuseMul}}$. This flavor of composition is standard (it is used, for example, in Doerner et al. [DKLS19]), and so we will not take pains to show it secure (though we intend to present an argument in the full version of this document). Note that we give the ideal adversary slightly more power than strictly necessary, in order to simplify our description: when it cheats, it always learns the secret inputs of *all* honest parties; in the real protocol, on the other hand the adversary may cheat on honest parties individually.

Functionality C.2. $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$**.** $\boldsymbol{n}$**-Party Multiplication**

This functionality is parameterized by the party count n and a prime modulus m. In addition to the parties it interacts directly with an ideal adversary $\mathcal{S}$ who corrupts the parties indexed by $\mathbf{P}^*$. The remaining honest parties are indexed by $\overline{\mathbf{P}^*} := [n] \setminus \mathbf{P}^*$.

Cheater Activation: Upon receiving $(\texttt{cheat}, \mathsf{sid})$ from $\mathcal{S}$, store $(\texttt{cheater}, \mathsf{sid})$ in memory and send any record of the form $(\texttt{value}, \mathsf{sid}, i, x_i)$ to $\mathcal{S}$. For the purposes of this functionality, we will consider session IDs to be fresh even when a $\texttt{cheater}$ record already exists in memory.

Input: Upon receiving $(\texttt{input}, \mathsf{sid}, x_i)$ from each party $\mathcal{P}_i$ for $i \in [n]$, if $0 \le x_i < m$ for all $i \in [n]$, then store $(\texttt{value}, \mathsf{sid}, i, x_i)$ in memory for each $i \in [n]$ and send $(\texttt{value-loaded}, \mathsf{sid})$ to all parties. If a record of the form $(\texttt{cheat}, \mathsf{sid})$ exists in memory, then send $(\texttt{value}, \mathsf{sid}, i, x_i)$ to $\mathcal{S}$ for each $i \in [n]$.

Multiplication: Upon receiving $(\texttt{multiply}, \mathsf{sid}_1, \mathsf{sid}_2, \mathsf{sid}_3)$ from each party $\mathcal{P}_i$ for $i \in \overline{\mathbf{P}^*}$ and $(\texttt{adv-multiply}, \mathsf{sid}_1, \mathsf{sid}_2, \mathsf{sid}_3, i, z_i)$ from $\mathcal{S}$ for each $i \in \mathbf{P}^*$,[a] if all three session IDs are agreed upon and sid_3 is fresh, and if no record of the form $(\texttt{cheater}, \mathsf{sid}_1)$ or $(\texttt{cheater}, \mathsf{sid}_2)$ exists in memory, and if records of the form $(\texttt{value}, \mathsf{sid}_1, i, x_i)$ and $(\texttt{value}, \mathsf{sid}_2, i, y_i)$ exist in memory for all $i \in [n]$, then sample $z_i \leftarrow \mathbb{Z}_m$ for $i \in \overline{\mathbf{P}^*}$ subject to

$$\sum_{i\in[n]} z_i \equiv \sum_{i\in[n]} x_i \cdot \sum_{i\in[n]} y_i \pmod m$$

If the previous conditions hold, but $(\texttt{cheater}, \mathsf{sid}_1)$ or $(\texttt{cheater}, \mathsf{sid}_2)$ exists in memory, then send $(\texttt{cheat-multiply}, \mathsf{sid}_1, \mathsf{sid}_2, \mathsf{sid}_3)$ to $\mathcal{S}$ and in response receive $(\texttt{cheat-product}, \mathsf{sid}_3, \{z_i\}_{i\in\overline{\mathbf{P}^*}})$ where $0 \le z_i < m$ for all i. Regardless, send $(\texttt{product}, \mathsf{sid}_3, z_i)$ to each party $\mathcal{P}_i$ for $i \in [n]$ as adversarially delayed private output.

Cheater Check: Upon receiving $(\texttt{check}, \mathsf{sid})$ from all parties, if a record of the form $(\texttt{cheat}, \mathsf{sid})$ exists in memory, then abort, informing all parties in an adversarially-delayed fashion. Otherwise, send $(\texttt{no-cheater}, \mathsf{sid})$ to both parties as adversarially-delayed private output. Regardless, refuse all future messages with this sid, except for the $\texttt{open}$ message.

Input Revelation: Upon receiving $(\texttt{open}, \mathsf{sid})$ from all parties, if a record of the form $(\texttt{cheat}, \mathsf{sid})$ exists in memory, then abort, informing all parties in an adversarially delayed fashion. Otherwise, for each record of the form $(\texttt{value}, \mathsf{sid}, i, x_i)$ in memory, send $(\texttt{opened-value}, \mathsf{sid}, i, x_i)$ to all parties as adversarially delayed output. Refuse all future messages with this sid.

[a] In the semi-honest setting, the adversary does not send these values to the functionality; instead the functionality samples the shares for corrupt parties just as it does for honest parties.

We defer an efficiency analysis of the protocol that realizes this functionality to the next subsection.

C.3 Augmented Multiplication

Finally, we describe a protocol π_{AugMul} that realizes $\mathcal{F}_{\mathsf{AugMul}}$ in the $(\mathcal{F}_{\mathsf{ReuseMul}},\mathcal{F}_{\mathsf{ComCompute}})$-hybrid model. It comprises five phases. Its Input, Multiplication, and Input Revelation phases essentially fall through to $\mathcal{F}_{\mathsf{ReuseMul}}$. Its Cheater Check phase falls through to the Cheater Check phase of $\mathcal{F}_{\mathsf{ReuseMul}}$, but also takes additional steps to securely evaluate an arbitrary predicate over the checked values, using generic MPC. Finally, it adds a Sampling phase, which samples pairs of nonzero values by running a sequence of $\mathcal{F}_{\mathsf{ReuseMul}}$ instructions.

Protocol C.3. $\pi_{\mathsf{AugMul}}(n)$. Augmented Multiplication

This protocol is parameterized by the number of parties n, and let there be in addition a statistical security parameter, s. The parties have access to the $\mathcal{F}_{\mathsf{ReuseMul}}$ and $\mathcal{F}_{\mathsf{ComCompute}}$ functionalities.

Input: Each party $\mathcal{P}_i$ for $i \in [n]$ begins with an input share x_i, a modulus m, and a fresh, agreed-upon session ID sid.

1. $\mathcal{P}_i$ sends $(\texttt{input}, \mathsf{sid}\|1, x_i)$ to $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$, and waits to receive $(\texttt{value-loaded}, \mathsf{sid}\|1)$ in response.
2. $\mathcal{P}_i$ stores $(\texttt{value}, \mathsf{sid}, 1, x_i, m)$ in memory.

Sampling: Each party $\mathcal{P}_i$ for $i \in [n]$ begins with a modulus m and a pair of fresh, agreed-upon session IDs, sid_1 and sid_2. The parties set $\mathsf{ctr} := 0$ and $\mathsf{sid} := \mathsf{sid}_1\|\mathsf{sid}_2$.

3. $\mathcal{P}_i$ samples three private random values, $(r_i, x_i, y_i) \leftarrow \mathbb{Z}_m^3$, and then loads them into $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$ by sending the messages $(\texttt{input}, \mathsf{sid}_1\|\mathsf{ctr}, x_i)$, $(\texttt{input}, \mathsf{sid}_2\|\mathsf{ctr}, y_i)$, and $(\texttt{input}, \mathsf{sid}\|\mathsf{ctr}\|\texttt{r}, r_i)$, waiting after each for confirmation.
4. $\mathcal{P}_i$ sends $(\texttt{multiply}, \mathsf{sid}_1\|\mathsf{ctr}, \mathsf{sid}_2\|\mathsf{ctr}, \mathsf{sid}\|\mathsf{ctr}\|\texttt{xy})$ to $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$, and receives $(\texttt{product}, \mathsf{sid}\|\mathsf{ctr}\|\texttt{xy}, z_i)$ in response.

5. $\mathcal{P}_i$ sends $(\texttt{input}, \mathsf{sid}\|\mathsf{ctr}\|\texttt{z}, z_i)$ to $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$, and waits for confirmation, after which it sends $(\texttt{multiply}, \mathsf{sid}\|\mathsf{ctr}\|\texttt{z}, \mathsf{sid}\|\mathsf{ctr}\|\texttt{r}, \mathsf{sid}\|\mathsf{ctr}\|\texttt{zr})$ to $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$, and receives $(\texttt{product}, \mathsf{sid}\|\mathsf{ctr}\|\texttt{zr}, \tilde{z}_i)$ in response.

6. $\mathcal{P}_i$ sends $\tilde{z}_i$ to all other parties, and in response it receives $\tilde{z}_j$ for $j \in [n] \setminus \{i\}$. $\mathcal{P}_i$ stores $(\texttt{value}, \mathsf{sid}_1, \mathsf{ctr}, x_i, m)$ and $(\texttt{value}, \mathsf{sid}_2, \mathsf{ctr}, y_i, m)$ in memory and takes (x_i, y_i, z_i) to be its outputs if and only if

$$\sum_{j\in[n]} \tilde{z}_j \not\equiv 0 \pmod{m}$$

Otherwise, $\mathcal{P}_i$ sends $(\texttt{open}, \mathsf{sid}_1\|\mathsf{ctr})$, $(\texttt{open}, \mathsf{sid}_2\|\mathsf{ctr})$, and $(\texttt{open}, \mathsf{sid}\|\mathsf{ctr}\|\texttt{r})$ to $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$. If $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$ aborts, then $\mathcal{P}_i$ aborts. If $\mathcal{P}_i$ receives x_j, y_j, and r_j for $j \in [n]$ from $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$ in response, and if

$$\left(\sum_{j\in[n]} x_j\right) \cdot \left(\sum_{j\in[n]} y_j\right) \cdot \left(\sum_{j\in[n]} r_j\right) \not\equiv 0 \pmod{m}$$

then $\mathcal{P}_i$ aborts. Otherwise, $\mathcal{P}_i$ increments ctr and begins again from Step 3.

Multiplication: Each party $\mathcal{P}_i$ for $i \in [n]$ begins with a pair of input session IDs, sid_1 and sid_2, and an output session ID, sid_3, such that records of the form $(\texttt{value}, \mathsf{sid}_1, \mathsf{ctr}_1, x_i, m)$ and $(\texttt{value}, \mathsf{sid}_2, \mathsf{ctr}_2, y_i, m)$ exist in memory with the same value of m.

7. $\mathcal{P}_i$ sends $(\texttt{multiply}, \mathsf{sid}_1\|\mathsf{ctr}_1, \mathsf{sid}_2\|\mathsf{ctr}_2, \mathsf{sid}_3)$ to $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$, and receives $(\texttt{product}, \mathsf{sid}_3, z_i)$ in response, taking z_i to be its output.

Predicate Cheater Check: Each party $\mathcal{P}_i$ for $i \in [n]$ begins with a vector of input session IDs $\mathbf{sids}$ such that for each $\mathsf{sid} \in \mathbf{sids}$, there exists a record of the form $(\texttt{value}, \mathsf{sid}, *, *, *)$ in that party's memory, and with the description of a predicate f over the stored values associated with the input session IDs. No party includes a single input session ID in more than one execution of this protocol phase, nor does any party include an input session ID in any execution of this phase if the Input Revelation phase has already been run with the same ID. Let s be the statistical security parameter, and for convenience, let $\mathsf{joint\text{-}sid}$ be a unique value derived from $\mathbf{sids}$, and let $\mathbf{m}$ be a vector (without duplication) of all the moduli associated with the records referenced by the IDs in $\mathbf{sids}$, and let $\mathsf{filter}(\mathbf{sids}, m)$ denote the subvector of IDs in $\mathbf{sids}$associated with records that have the modulus m.

8. For each $m \in \mathbf{m}$, in parallel:

(a) For each $\mathsf{sid} \in \mathsf{filter}(\mathsf{\textbf{sids}}, m)$, each party $\mathcal{P}_i$ for $i \in [n]$ retrieves its record $(\texttt{value}, \mathsf{sid}, *, x_i, m)$ from memory and sends $(\texttt{commit}, \mathsf{joint\text{-}sid}\|\texttt{val}\|\mathsf{sid}\|i, x_i)$ to $\mathcal{F}_{\mathsf{ComCompute}}(n)$, waiting afterward for confirmation that all parties have submitted inputs.

(b) Each party $\mathcal{P}_i$ samples a vector $\mathbf{r}_{i,*} \leftarrow \mathbb{Z}_m^{\lceil s/|m|\rceil}$.

(c) For each $j \in [\lceil s/|m|\rceil]$, each party $\mathcal{P}_i$ for $i \in [n]$ sends $(\texttt{commit}, \mathsf{joint\text{-}sid}\|\texttt{key}\|m\|j\|i, \mathbf{r}_{i,j})$ to $\mathcal{F}_{\mathsf{ComCompute}}(n)$ and $(\texttt{input}, \mathsf{joint\text{-}sid}\|\texttt{key}\|j, \mathbf{r}_{i,j})$ to $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$, waiting afterward for confirmation that all other parties have submitted their inputs to both functionalities.

(d) For each $j \in [\lceil s/|m|\rceil]$ and each $\mathsf{sid} \in \mathsf{filter}(\mathsf{\textbf{sids}}, m)$, in parallel:

   i. Each $\mathcal{P}_i$ retrieves its record $(\texttt{value}, \mathsf{sid}, \mathsf{ctr}, x_i, m)$ and sends $(\texttt{multiply}, \mathsf{sid}\|\mathsf{ctr}, \mathsf{joint\text{-}sid}\|\texttt{key}\|j, \mathsf{joint\text{-}sid}\|\texttt{tag}\|j)$ to $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$, and receives $(\texttt{product}, \mathsf{joint\text{-}sid}\|\texttt{tag}\|j, \mathbf{t}_{i,j})$ in response.

   ii. Each $\mathcal{P}_i$ sends $(\texttt{commit}, \mathsf{joint\text{-}sid}\|\texttt{tag}\|m\|j\|i, \mathbf{t}_{i,j})$ to $\mathcal{F}_{\mathsf{ComCompute}}(n)$, waiting afterward for confirmation that all parties have submitted inputs.

   iii. Let f' be a description of the circuit that verifies whether

$$\left(\sum_{i\in[n]} \mathbf{r}_{i,j}\right) \cdot \left(\sum_{i\in[n]} x_i\right) \equiv \sum_{i\in[n]} \mathbf{t}_{i,j} \pmod{m}$$

   and let

$$\mathsf{\textbf{check-sids}} := \left\{ \begin{matrix} \mathsf{joint\text{-}sid}\|\texttt{key}\|m\|j\|i', \\ \mathsf{joint\text{-}sid}\|\texttt{val}\|\mathsf{sid}\|i', \\ \mathsf{joint\text{-}sid}\|\texttt{tag}\|m\|j\|i' \end{matrix} \right\}_{i'\in[n]}$$

   Each party $\mathcal{P}_i$ sends $(\texttt{compute}, \mathsf{sid}\|j, \mathsf{\textbf{check-sids}}, f')$ to $\mathcal{F}_{\mathsf{ComCompute}}(n)$, and aborts if $\mathcal{F}_{\mathsf{ComCompute}}(n)$ aborts or if $\mathcal{F}_{\mathsf{ComCompute}}(n)$ indicates that the predicate f' is false.

(e) For each $j \in [\lceil s/|m|\rceil]$, each party $\mathcal{P}_i$ for $i \in [n]$ sends $(\texttt{check}, \mathsf{joint\text{-}sid}\|\texttt{key}\|j)$ to $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$, and aborts if $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$ aborts.

(f) For each $\mathsf{sid} \in \mathsf{filter}(\mathsf{\textbf{sids}}, m)$, each party $\mathcal{P}_i$ for $i \in [n]$ retrieves its record $(\texttt{value}, \mathsf{sid}, \mathsf{ctr}, x_i, m)$ and sends $(\texttt{check}, \mathsf{sid}\|\mathsf{ctr})$ to $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$. If $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$ aborts, then $\mathcal{P}_i$ aborts.

9. Let

$$\mathsf{\textbf{pred-sids}} := \{\mathsf{joint\text{-}sid}\|\texttt{val}\|\mathsf{sid}\|i'\}_{i'\in[n],\mathsf{sid}\in\mathsf{\textbf{sids}}}$$

Each party $\mathcal{P}_i$ sends $(\texttt{compute}, \textsf{joint-sid}\|\texttt{predicate}, \textbf{pred-sids}, f)$ to $\mathcal{F}_{\mathsf{ComCompute}}(n)$ and aborts if $\mathcal{F}_{\mathsf{ComCompute}}(n)$ aborts or if $\mathcal{F}_{\mathsf{ComCompute}}(n)$ indicates that the predicate f is false. If $\mathcal{F}_{\mathsf{ComCompute}}(n)$ indicates that the predicate is true, then $\mathcal{P}_i$ takes the Predicate Cheater Check to have been successful.

Input Revelation: Each party $\mathcal{P}_i$ for $i \in [n]$ begins with a fresh session ID sid, such that a record of the form $(\texttt{value}, \mathsf{sid}, \mathsf{ctr}, *, m)$ exists in that party's memory. No party executes this phase with the same sid more than once.

10. $\mathcal{P}_i$ sends $(\texttt{open}, \mathsf{sid}\|\mathsf{ctr})$ to $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$, and then waits to receive $(\texttt{opened-value}, \mathsf{sid}\|\mathsf{ctr}, j, x_j)$ from $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$ for $j \in [n]$. It then takes x_j for $j \in [n]$ to be its outputs.

We now discuss security and efficiency phase-by-phase.

Input. The input phase of π_{AugMul} defers directly to $\mathcal{F}_{\mathsf{ReuseMul}}$, and therefore inherits its security. When realized as we have discussed in Section C.2, a single call to $\mathcal{F}_{\mathsf{ReuseMul}}$ among all parties corresponds to all pairs of parties making two calls each to $\mathcal{F}_{\mathsf{ReuseMul2P}}$. Recall that in $\mathcal{F}_{\mathsf{ReuseMul2P}}$, loading inputs from the party playing Bob is effectively free, and as a consequence, we need only count costs due to inputs loaded from Alice. The first party, $\mathcal{P}_1$, plays Alice in all of its interactions with $\mathcal{F}_{\mathsf{ReuseMul2P}}$, and pays a cost of $(n-1)\cdot|m|\cdot(|m|+2s)$ bits if $\mathcal{F}_{\mathsf{DelayedCOT}}$ is realized via Silent OT [BCG+19] or KOS OTe [KOS15]. The last party, $\mathcal{P}_n$, always plays Bob, and pays a cost of $(n-1)\cdot|m|$ bits if $\mathcal{F}_{\mathsf{DelayedCOT}}$ is realized via Silent OT, or $(n-1)\cdot 2s\cdot(|m|+2s)$ bits if $\mathcal{F}_{\mathsf{DelayedCOT}}$ is realized via KOS OTe. The other parties play a mixture of the roles, and pay an average cost of

$$(n-1)\cdot|m|\cdot\frac{(|m|+2s+1)}{2} \qquad \text{or} \qquad (n-1)\cdot\frac{(|m|+2s)^2}{2}$$

bits with Silent OT and KOS OTe respectively. The former realization requires two rounds, and the latter requires three.

Multiplication. The input phase of π_{AugMul} defers directly to $\mathcal{F}_{\mathsf{ReuseMul}}$. As we have noted in Section C.1, the multiplication and input phases of $\mathcal{F}_{\mathsf{ReuseMul2P}}$ cost the same; however, whereas the input phase of π_{AugMul} corresponds costwise to one invocation of the input phase of $\mathcal{F}_{\mathsf{ReuseMul2P}}$ for each pair of parties (due to Bob's inputs being free), the multiplication phase of π_{AugMul} corresponds to *two* invocations of the multiplication phase of $\mathcal{F}_{\mathsf{ReuseMul2P}}$ for each pair of parties. Thus the parties pay an average cost of

$$(n-1)\cdot|m|\cdot(|m|+2s+1) \qquad \text{or} \qquad (n-1)\cdot(|m|+2s)^2$$

bits with Silent OT and KOS OTe respectively, and two or three rounds. Note, that as an optimization two input phases can be fused with one multiplication (in which they are used), and the inputs will consequently add no additional cost.

Input Revelation. The input revelation phase of π_{AugMul} defers directly to $\mathcal{F}_{\mathsf{ReuseMul}}$, and corresponds to two invocations of the $\mathcal{F}_{\mathsf{ReuseMul2P}}$ open command for each pair of parties: one invocation on Alice's part, and one on Bob's. Thus the cost of this phase is

$$(n-1)\cdot\frac{(|m|+1)\cdot(|m|+2s)}{2}$$

bits per party on average, and three messages.

Sampling. This procedure is probabilistic. Specifically, each iteration succeeds with probability $((m-1)/m)^3$, and thus $(m/(m-1))^3$ iterations are required in expectation. Each iteration requires two calls to the $\mathcal{F}_{\mathsf{ReuseMul}}$ multiplication command (we will assume that the $\mathcal{F}_{\mathsf{ReuseMul}}$ input command is coalesced and therefore free, as described previously), and all iterations after the first require two invocations of the $\mathcal{F}_{\mathsf{ReuseMul}}$ open command. Thus, if we use Silent OT to realize $\mathcal{F}_{\mathsf{DelayedCOT}}$, then the average cost per party due to multiplication is

$$\left(\frac{m}{m-1}\right)^3\cdot(n-1)\cdot 2|m|\cdot(|m|+2s+1)$$

bits and $5\cdot(m/(m-1))^3$ rounds, whereas with KOS it is

$$\left(\frac{m}{m-1}\right)^3\cdot(n-1)\cdot 2\cdot(|m|+2s)^2$$

bits and $7\cdot(m/(m-1))^3$ rounds, and in either case the parties must also pay

$$\left(\left(\frac{m}{m-1}\right)^3-1\right)\cdot(n-1)\cdot(|m|+1)\cdot(|m|+2s)$$

bits and $3\cdot((m/m-1)^3-1)$ rounds on average due to openings of values. For values of m of any significant size, only one iteration will be required in expectation, and these costs converge to the cost of two sequential multiplications, plus one additional round.

With respect to security, we observe that the values $\tilde{z}_i$ for $i\in[n]$ jointly reveal nothing about the secret values x_i and y_i, because the latter pair of values have been masked by r_i. Thus the security of a successful iteration reduces direction to the security of the constituent multipliers; that is, successful iterations admit cheats, which can later be caught in the Cheater Check phase. In failed iterations, all values are opened and the computations are checked locally by each party. This ensures that the adversary cannot force sampling failures by cheating, and thereby prevent the protocol from terminating.

Predicate Cheater Check. Unlike the other protocols, this protocol takes an input of flexible dimension and therefore does not have a convenient closed-form cost. Consequently we will describe the cost piecemeal. For each input to be checked, let m be the modulus with which the input is associated and let c be the number of multiplications in which it has been used. The parties engage in $s/|m|$ additional forced-reuse multiplications, with costs as previously described. They then run the Cheater Check command of $\mathcal{F}_{\mathsf{ReuseMul2P}}$ in a pairwise fashion, incurring an additional cost of

$$(n-1) \cdot s \cdot (|m| + 2s) \cdot (c + s/|m|)$$

bits transmitted per party on average, and one message. Finally, they each input $|m| + s$ bits into a generic MPC, and then run a circuit that performs $3n \cdot s/|m|$ modular additions and $s/|m|$ modular multiplications and equality tests over $\mathbb{Z}_m$. The size of this circuit is roughly $(12n + 1) \cdot s + 8s \cdot |m|$ AND gates.[8] In addition to these costs for *each* input to be checked, the generic MPC also evaluates the predicate f, comprising $|f|$ AND gates, over the inputs already loaded.

With respect to security, we note that the protocol effectively uses a straightforward composition of secure parts to implement an information-theoretic MAC over the shared values corresponding to the inputs to be checked, in order to ensure that they are transferred into the circuit of the generic MPC faithfully. Forced reuse ensures that the MACs are applied to the correct values, and because each MAC has soundness error $1/m$, it is necessary to repeat the process $s/|m|$ times. The multiplications (including those used to apply the MACs) are then checked for cheats, and the MACs are verified inside the circuit before the predicate f is evaluated.

D Proof of Security for Distributed Modulus Sampling

In this section, we provide the full proof of Theorem 4.6, showing that π_{RSAGen} realizes $\mathcal{F}_{\mathsf{RSAGen}}$ in the malicious setting.

Theorem 4.6. *Assume that factoring is hard with respect to* BFGM. *Then, protocol* π_{RSAGen} *UC-realizes* $\mathcal{F}_{\mathsf{RSAGen}}$ *in the* $(\mathcal{F}_{\mathsf{AugMul}}, \mathcal{F}_{\mathsf{Biprime}})$*-hybrid model against a static, malicious adversary that corrupts up to* $n-1$ *parties.*

Proof. We begin by describing a simulator $\mathcal{S}_{\mathsf{RSAGen}}$ for the dummy adversary $\mathcal{A}$. Next, we prove by a sequence of hybrid experiments that no PPT environment can distinguish between running with the dummy adversary and real parties executing π_{RSAGen} from running with $\mathcal{S}_{\mathsf{RSAGen}}$ and dummy parties interacting with $\mathcal{F}_{\mathsf{RSAGen}}$, with more than negligible probability.

[8] We take the gate size of a modular addition in $\mathbb{Z}_m$ to be $4|m|$ and the gate size of a modular doubling to be $3|m|$, and from these we derive the gate size of a modular multiplication to be $8|m|^2$.

Simulator D.1. $\mathcal{S}_{\mathsf{RSAGen}}(\kappa, n, \mathbf{P}^*)$. Distributed Modulus Sampling

Let $\mathbf{P}^*$ be the set of corrupted parties, and let $\mathbf{m} = (\mathbf{m}_1, \ldots, \mathbf{m}_{\ell'})$ and $\ell < \ell'$ be as defined in π_{RSAGen} for κ and n. The simulator initializes two flags: `not_biprime` $= 0$ and $\mathsf{cheatflag} = 0$; if at any point during the protocol the adversary sends $(\texttt{cheat}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{AugMul}}$, then set $\mathsf{cheatflag} = 1$.

Candidate Sieving:

1. To simulate Step 1, receive $(\texttt{adv-sample}, \mathsf{psids}_j, \mathsf{qsids}_j, \mathbf{p}_{i,j}, \mathbf{q}_{i,j}, \mathbf{N}_{i,j}, \mathbf{m}_j)$ from $\mathcal{A}$ for every $j \in [2, \ell]$ and on behalf of every corrupted party $\mathcal{P}_i$.

   - In case $\mathsf{cheatflag} = 1$ and $\mathcal{A}$ sent $(\texttt{cheat}, \mathsf{psids}_j)$ or $(\texttt{cheat}, \mathsf{qsids}_j)$ to $\mathcal{F}_{\mathsf{AugMul}}$, send $(\texttt{cheat-sample}, \mathsf{psids}_j, \mathsf{qsids}_j)$ to $\mathcal{A}$ and receive back $(\texttt{cheat-product}, \mathsf{psids}_j, \mathsf{qsids}_j, \{(\mathbf{p}_{i,j}, \mathbf{q}_{i,j}, \mathbf{N}_{i,j})\}_{i \in \overline{\mathbf{P}^*}})$.
   - Otherwise, if $\mathsf{cheatflag} = 1$ but $\mathcal{A}$ did not send $(\texttt{cheat}, \mathsf{psids}_j)$ and $(\texttt{cheat}, \mathsf{qsids}_j)$ to $\mathcal{F}_{\mathsf{AugMul}}$, for every $i \in \overline{\mathbf{P}^*}$ sample uniformly random values $\mathbf{p}_{i,j}, \mathbf{q}_{i,j}, \mathbf{N}_{i,j} \leftarrow \mathbb{Z}_{\mathbf{m}_j}$ condition on

   $$\sum_{i \in [n]} \mathbf{N}_{i,j} \equiv \sum_{i \in [n]} \mathbf{p}_{i,j} \cdot \sum_{i \in [n]} \mathbf{q}_{i,j} \not\equiv 0 \pmod{\mathbf{m}_j}$$

   Finally, send $(\texttt{sampled-product}, \mathsf{psids}_j, \mathsf{qsids}_j, \mathbf{p}_{i,j}, \mathbf{q}_{i,j}, \mathbf{N}_{i,j})$ to $\mathcal{A}$ for every $i \in \mathbf{P}^*$.

2. For each $i \in \mathbf{P}^*$, set $\mathbf{p}_{i,1} = \mathbf{q}_{i,1} = 3$ if $i = 1$, or 0 otherwise, and compute

   $$p_i := \mathsf{CRTRecon}\left(\{\mathbf{m}_j\}_{j \in [\ell]}, \{\mathbf{p}_{i,j}\}_{j \in [\ell]}\right)$$
   $$q_i := \mathsf{CRTRecon}\left(\{\mathbf{m}_j\}_{j \in [\ell]}, \{\mathbf{q}_{i,j}\}_{j \in [\ell]}\right)$$

   Denote $p_{\mathcal{A}} = \sum_{i \in \mathbf{P}^*} p_i$ and $q_{\mathcal{A}} = \sum_{i \in \mathbf{P}^*} q_i$. If $\mathsf{cheatflag} = 1$, compute for every $i \in \overline{\mathbf{P}^*}$ the values p_i and q_i in a similar way.

3. To simulate Step 3, for every $j \in [\ell + 1, \ell']$:

   - Receive $(\texttt{input}, \mathsf{psids}_j, \mathbf{p}_{i,j}, \mathbf{m}_j)$ followed by $(\texttt{input}, \mathsf{qsids}_j, \mathbf{q}_{i,j}, \mathbf{m}_j)$ from $\mathcal{A}$ on behalf of every corrupted party $\mathcal{P}_i$ and answer accordingly with $(\texttt{value-loaded}, \mathsf{sid})$ as $\mathcal{F}_{\mathsf{AugMul}}$ would.
   - In case $\mathsf{cheatflag} = 1$, for every $i \in \overline{\mathbf{P}^*}$:
     (a) Compute $\mathbf{p}_{i,j} := p_i \bmod \mathbf{m}_j$ and $\mathbf{q}_{i,j} := q_i \bmod \mathbf{m}_j$.
     (b) If $\mathcal{A}$ sent $(\texttt{cheat}, \mathsf{psids}_j)$ send $(\texttt{value}, \mathsf{psids}_j, i, \mathbf{p}_{i,j}, \mathbf{m}_j)$, and if $\mathcal{A}$ sent $(\texttt{cheat}, \mathsf{qsids}_j)$ send $(\texttt{value}, \mathsf{qsids}_j, i, \mathbf{q}_{i,j}, \mathbf{m}_j)$.
   - Next, receive $(\texttt{multiply}, \mathsf{psids}_j, \mathsf{qsids}_j, \mathsf{Nsids}_j)$ from $\mathcal{A}$ along with $(\texttt{adv-multiply}, \mathsf{psids}_j, \mathsf{qsids}_j, \mathsf{Nsids}_j, i, \mathbf{N}_{i,j})$ for every corrupted $\mathcal{P}_i$.

- In case $\mathsf{cheatflag} = 1$:
  (a) If $\mathcal{A}$ sent $(\texttt{cheat}, \mathsf{psids}_j)$ or $(\texttt{cheat}, \mathsf{qsids}_j)$ to $\mathcal{F}_{\mathsf{AugMul}}$, send $(\texttt{cheat-multiply}, \mathsf{psids}_j, \mathsf{qsids}_j)$ to $\mathcal{A}$ and receive back $(\texttt{cheat-product}, \mathsf{psids}_j, \mathsf{qsids}_j, \{\mathbf{N}_{i,j}\}_{i\in\overline{\mathbf{P}^*}})$.
  (b) Otherwise, for every $i \in \overline{\mathbf{P}^*}$ sample $\mathbf{N}_{i,j} \leftarrow \mathbb{Z}_{\mathbf{m}_j}$ such that

$$\sum_{i\in[n]} \mathbf{N}_{i,j} \equiv \sum_{i\in[n]} \mathbf{p}_{i,j} \cdot \sum_{i\in[n]} \mathbf{q}_{i,j} \pmod{\mathbf{m}_j}.$$

- Finally, send $(\texttt{product}, \mathsf{Nsids}_j, \mathbf{N}_{i,j})$ to $\mathcal{A}$ for every $i \in \mathbf{P}^*$.

4. If $\exists j \in [\ell+1, \ell']$ and $i \in \mathbf{P}^*$ such that $\mathbf{p}_{i,j} \not\equiv p_i \pmod{\mathbf{m}_j}$ or $\mathbf{q}_{i,j} \not\equiv q_i \pmod{\mathbf{m}_j}$, then set $\mathsf{cheatflag} = 1$.

5. If $\mathsf{cheatflag} = 1$, send $(\texttt{abort}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{RSAGen}}$. Otherwise, if $\mathsf{cheatflag} = 0$, send $(\texttt{adv-input}, \mathsf{sid}, i, p_i, q_i)$ to $\mathcal{F}_{\mathsf{RSAGen}}$ for every $i \in \mathbf{P}^*$ and receive either $(\texttt{factors}, \mathsf{sid}, p, q)$ or $(\texttt{biprime}, \mathsf{sid}, N)$ in response. If received $(\texttt{factors}, \cdot)$ then set $\texttt{not\_biprime} = 1$ and compute $N = p \cdot q$. In either case, set $N_j := N \bmod \mathbf{m}_j$ for every $j \in [\ell']$.

6. To simulate Step 4:

   - If $\mathsf{cheatflag} = 1$, then the values $\mathbf{N}_{i,j}$ have already been determined for every $i \in \overline{\mathbf{P}^*}$ and $j \in [\ell']$.
   - If $\mathsf{cheatflag} = 0$, then the values $\mathbf{N}_{i,j}$ have been set only for corrupted parties. For every $i \in \overline{\mathbf{P}^*}$ and $j \in [\ell']$ sample uniformly distributed $\mathbf{N}_{i,j} \leftarrow \mathbb{Z}_{\mathbf{m}_j}$ such that

$$\sum_{i\in[n]} \mathbf{N}_{i,j} \equiv N_j \pmod{\mathbf{m}_j}$$

   For every $j \in [\ell']$ simulate broadcasting $\mathbf{N}_{i,j}$ to $\mathcal{A}$ on behalf of every honest party, and receiving $\mathbf{N}'_{i,j}$ from $\mathcal{A}$ for every corrupted party. If for some $i \in \mathbf{P}^*$ it holds that $\mathbf{N}'_{i,j} \neq \mathbf{N}_{i,j}$, set $\mathsf{cheatflag} = 1$.

7. Given the values $\mathbf{N}'_{i,j}$ received from $\mathcal{A}$ for $i \in \mathbf{P}^*$ and the values $\mathbf{N}'_{i,j} := \mathbf{N}_{i,j}$ simulated for $i \in \overline{\mathbf{P}^*}$, compute

$$N' := \sum_{i\in[n]} \mathsf{CRTRecon}\left(\mathbf{m}, \mathbf{N}'_{i,*}\right)$$

   (Note that $N' = N$ if $\mathsf{cheatflag} = 0$.)

8. To simulate Step 5, try to factor N' by all primes smaller than B. If N' is factored, proceed to simulating the consistency check.

Biprimality Test:

9. To simulate Step 6 receive $(\texttt{check-biprimality}, \mathsf{sid}, N', p_i^*, q_i^*)$ from $\mathcal{A}$ on behalf of every corrupted $\mathcal{P}_i$. Let $p_{\mathcal{A}}^* = \sum_{i \in \mathbf{P}^*} p_i^*$ and $q_{\mathcal{A}}^* = \sum_{i \in \mathbf{P}^*} qs_i$. If $p_{\mathcal{A}}^* \neq p_{\mathcal{A}}$ or $p_{\mathcal{A}}^* \neq p_{\mathcal{A}}$ set $\mathsf{cheatflag} = 1$.

10. If $\mathcal{A}$ sends $(\texttt{leak-shares}, \mathsf{sid})$, set $\mathsf{cheatflag} = 1$ and proceed as follows:

    - If the values p_i and q_i have been set in Step 2 of the simulation for the honest parties (i.e., if $\mathsf{cheatflag} = 1$ when simulating Step 2), denote $p_i^* \coloneqq p_i$ and $q_i^* \coloneqq q_i$ for every $i \in \overline{\mathbf{P}}^*$.
    - Otherwise, send $(\texttt{cheat}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{RSAGen}}$ and receive $(\texttt{factors}, \mathsf{sid}, p, q)$ in response. Compute for every $i \in \overline{\mathbf{P}}^*$ values p_i^* and q_i^* such that

    $$\sum_{i \in \overline{\mathbf{P}}^*} p_i^* = p - p_{\mathcal{A}} \qquad \text{and} \qquad \sum_{i \in \overline{\mathbf{P}}^*} q_i^* = q - q_{\mathcal{A}}$$

    Send $(\texttt{leaked-shares}, \mathsf{sid}, \{(p_i^*, q_i^*)\}_{i \in [n]})$ to $\mathcal{A}$ followed by $(\texttt{not-biprime}, \mathsf{sid})$ for every corrupted party.

11. If $\mathcal{A}$ did *not* send $(\texttt{leak-shares}, \mathsf{sid})$:

    - If $\mathsf{cheatflag} = 1$ or $\texttt{not\_biprime} = 1$, then send $(\texttt{not-biprime}, \mathsf{sid})$ to $\mathcal{A}$ for every corrupted party.
    - If $\mathsf{cheatflag} = 0$ and $\texttt{not\_biprime} = 0$, then send $(\texttt{biprime}, \mathsf{sid})$ to $\mathcal{A}$ for every corrupted party.

Consistency Check:

12. To simulate Step 7, proceed as follows

    - If $\mathsf{cheatflag} = 0$ and $\texttt{not\_biprime} = 0$, receive $(\texttt{check}, \mathsf{psids} \| \mathsf{qsids}, f)$ from $\mathcal{A}$ on behalf of every corrupted party and respond with $(\texttt{predicate-result}, \mathsf{psids} \| \mathsf{qsids}, 1)$.
    - If $\mathsf{cheatflag} = 1$ or $\texttt{not\_biprime} = 1$, for every $j \in [2, \ell']$ receive $(\texttt{open}, \mathsf{psids}_j)$ and $(\texttt{open}, \mathsf{qsids}_j)$ from $\mathcal{A}$ on behalf of every corrupted party. In case $(\texttt{cheat}, \mathsf{sid})$ was not send to $\mathcal{F}_{\mathsf{RSAGen}}$ in Step 10 of the simulation, send $(\texttt{cheat}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{RSAGen}}$, receive $(\texttt{factors}, \mathsf{sid}, p, q)$ in response, and compute for every $i \in \overline{\mathbf{P}}^*$ values p_i^* and q_i^* such that

    $$\sum_{i \in \overline{\mathbf{P}}^*} p_i^* = p - p_{\mathcal{A}} \qquad \text{and} \qquad \sum_{i \in \overline{\mathbf{P}}^*} q_i^* = q - q_{\mathcal{A}}$$

    Next, for $j \in [\ell']$ set $\mathbf{p}_{i,j} \coloneqq p_i^* \bmod \mathbf{m}_j$ and $\mathbf{q}_{i,j}) \coloneqq q_i^* \bmod \mathbf{m}_j$. Finally, respond with $(\texttt{opening}, \mathsf{psids}_j, \mathbf{p}_{i,j})$ and $(\texttt{opening}, \mathsf{qsids}_j, \mathbf{q}_{i,j})$ for each $i \in [n]$ and $j \in [2, \ell']$.

We now turn to prove that no PPT environment can distinguish with more than negligible probability between the execution of the protocol π_{RSAGen} with the dummy adversary $\mathcal{A}$ from the ideal computation with $\mathcal{F}_{\mathsf{RSAGen}}$ and $\mathcal{S}_{\mathsf{RSAGen}}$, by defining a sequence of hybrid experiment. The output of each experiment is the output of the environment. Let $\mathcal{Z}$ be an environment.

Hybrid $\mathcal{H}_1$. In this hybrid experiment, the simulator has access to the internal state of $\mathcal{F}_{\mathsf{RSAGen}}$: it can see the messages sent by the honest parties and choose their output values. The simulator emulates the honest parties in the protocol π_{RSAGen}, as well as $\mathcal{F}_{\mathsf{AugMul}}$ and $\mathcal{F}_{\mathsf{Biprime}}$, towards the adversary $\mathcal{A}$ and sets the output for each honest party according to its output in the simulation. Clearly, the output of $\mathcal{H}_1$ is identically distributed as $\textsc{real}_{\pi_{\mathsf{RSAGen}},\mathcal{A},\mathcal{Z}}$.

Hybrid $\mathcal{H}_2$. This hybrid experiment follows as $\mathcal{H}_1$ except that when the adversary sends the corrupted parties' inputs to $\mathcal{F}_{\mathsf{Biprime}}$, $(\texttt{check-biprimality}, \mathsf{sid}, N', p_i^*, q_i^*)$ for $i \in \mathbf{P}^*$, the simulator computes $p_{\mathcal{A}}^* = \sum_{i\in\mathbf{P}^*} p_i^*$ and $q_{\mathcal{A}}^* = \sum_{i\in\mathbf{P}^*} qs_i$, and checks whether $p_{\mathcal{A}}^* \neq p_{\mathcal{A}}$ or $p_{\mathcal{A}}^* \neq p_{\mathcal{A}}$ (where $p_{\mathcal{A}}$ and $q_{\mathcal{A}}$ are set according to $\mathcal{A}$'s inputs to $\mathcal{F}_{\mathsf{AugMul}}$). In case $p_{\mathcal{A}}^* \neq p_{\mathcal{A}}$ or $p_{\mathcal{A}}^* \neq p_{\mathcal{A}}$ the simulator returns $(\texttt{not-biprime}, \mathsf{sid})$ as the output of $\mathcal{F}_{\mathsf{Biprime}}$.

Claim D.2. *Assuming that factoring is hard with respect to* BFGM*, the output of $\mathcal{H}_2$ is computationally indistinguishable from the output of $\mathcal{H}_1$.*

Proof. Let $\mathcal{Z}$ be a PPT environment that can distinguish between $\mathcal{H}_2$ and $\mathcal{H}_1$ with probability ϵ. We will construct an expected polynomial time algorithm that succeeds in the factoring experiment (Experiment 3.1) with respect to BFGM with probability $\epsilon/2^7$. We start by defining a variant of the factoring experiment that receives as inputs arbitrary "shares" for p and for q from the adversary.

Experiment D.3. $\mathsf{Factor}'_{\mathcal{A},\mathbf{P}^*,\mathsf{GenModulus}'}(\kappa)$

1. Invoke $\mathcal{A}$ with 1^κ and receive back p_i and q_i for every $i \in \mathbf{P}^*$.
2. Run $(N, p, q) \leftarrow \mathsf{GenModulus}'(1^\kappa, \{(p_i, q_i)\}_{i\in\mathbf{P}^*})$.
3. Send N to $\mathcal{A}$, and receive $p', q' > 1$ in return.
4. Output 1 if and only if $p' \cdot q' = N$.

This experiment follows [HMR+19] that adjusted the standard factoring experiment from its semi-honest version, where the adversary has no influence on the distribution of p and q, to the malicious setting, where it can arbitrarily choose some shares. We recalled their lemma as Lemma 3.5.

To prove the claim we construct a reduction from the environment $\mathcal{Z}$ to an adversary $\mathcal{A}$ for Factor' as follows:

1. Invoke $\mathcal{Z}$ and simulate Steps 1-3 of π_{RSAGen} (this is identical both in $\mathcal{H}_1$ and in $\mathcal{H}_2$). Using the inputs of $\mathcal{Z}$ to $\mathcal{F}_{\mathsf{AugMul}}$, construct the adversarial

inputs $(p_i, q_i)_{i \in \mathbf{P}^*}$ and forward them to the challenger. Let $p_{\mathcal{A}} = \sum_{i \in \mathbf{P}^*} p_i$ and $q_{\mathcal{A}} = \sum_{i \in \mathbf{P}^*} q_i$.

2. Upon receiving N from the challenger, compute $\mathbf{N}_{i,j}$ for $i \in \overline{\mathbf{P}^*}$ and $j \in [\ell']$ such that $\sum_i \mathbf{N}_{i,j} \equiv N \pmod{\mathbf{m}_j}$ (the values $\mathbf{N}_{i,j}$ for $i \in \overline{\mathbf{P}^*}$ and $j \in [\ell']$ are set by $\mathcal{Z}$'s inputs to $\mathcal{F}_{\mathsf{AugMul}}$).

3. Broadcast $\mathbf{N}_{i,j}$ for honest parties and receive $\mathbf{N}'_{i,j}$ for corrupted parties. Reconstruct $\mathbf{N}'$ from these values (as done in Step 4 of π_{RSAGen}).

4. If $\mathbf{N}'$ is factor by a primes smaller than B, then abort.

5. Receive p_i^* and q_i^* from each corrupted $\mathcal{P}_i$ from $\mathcal{Z}$ as inputs to $\mathcal{F}_{\mathsf{Biprime}}$.

6. Let $p_{\mathcal{A}}^* = \sum_{i \in \mathbf{P}^*} p_i^*$ and $q_{\mathcal{A}}^* = \sum_{i \in \mathbf{P}^*} q_i^*$. If $p_{\mathcal{A}} = p_{\mathcal{A}}^*$, abort.

7. Try to solve the following system of equations for unknowns $p_{\mathcal{H}}$ and $q_{\mathcal{H}}$

$$(p_{\mathcal{A}}^* + p_{\mathcal{H}}) \cdot (q_{\mathcal{A}}^* + q_{\mathcal{H}}) = N \quad \text{and} \quad (p_{\mathcal{A}}^* + q_{\mathcal{H}}) \cdot (q_{\mathcal{A}}^* + p_{\mathcal{H}}) = N$$

   If so, then find p_h, q_h as the solution to the above two equations and output these values. This can be done by expressing q_h in terms of p_h and taking the positive root of the resulting quadratic.

Assuming $\mathtt{cheatflag} = \mathtt{not\_biprime} = 0$ the view of $\mathcal{A}$ in the above protocol (up to querying $\mathcal{F}_{\mathsf{Biprime}}$) is identical to its view in both hybrids $\mathcal{H}_1$ and $\mathcal{H}_2$, by perfect security of the additive sharing $\mathbf{N}$. If it induces the distinguishing event bad then it holds that $p_{\mathcal{A}} - p_{\mathcal{A}}' \neq 0$ and $q_{\mathcal{A}} - q_{\mathcal{A}}' \neq 0$. This means that Step 7 always (when bad occurs) computes p_h, q_h as per the constraints, which directly yields the factorization of N. Therefore $\Pr[\mathsf{Expt}_{\mathsf{BF}}^{\mathsf{A}} = 1] = \Pr[\mathsf{bad}] = \epsilon$. Finally applying Lemma 3.5, we have that if there is a poly-time distinguisher for experiments $\mathcal{H}_2$ and $\mathcal{H}_1$ successful with probability ϵ, then there is an expected poly-time adversary for the factoring problem (i.e., $\mathsf{Factor}_{\mathcal{A},\mathsf{BFGM}}(\kappa)$) that succeeds with probability $\epsilon/2^4 n^3$. □

Hybrid $\mathcal{H}_3$. This hybrid experiment follows as $\mathcal{H}_2$ except for the following differences:

- The simulator starts simulating Steps 1-3 of π_{RSAGen} without running the code of the honest parties, and simply communicates with $\mathcal{A}$ as $\mathcal{F}_{\mathsf{AugMul}}$ would.

- If $\mathcal{A}$ sends a $(\mathtt{cheat}, \cdot)$ message at some point, the simulator retrospectively simulates the honest parties according to the real protocol and answers the $\mathtt{cheat}$ request according to $\mathcal{H}_2$. The simulator continues the simulation exactly as $\mathcal{H}_2$.

- If $\mathcal{A}$ does not send a $(\mathtt{cheat}, \cdot)$ message to $\mathcal{F}_{\mathsf{AugMul}}$ during Steps 1-3, for each $i \in \mathbf{P}^*$, the simulator sets $\mathbf{p}_{i,1} = \mathbf{q}_{i,1} = 3$ if $i = 1$, or 0 otherwise, and computes

$$p_i := \mathsf{CRTRecon}\big(\{\mathbf{m}_j\}_{j\in[\ell]}, \{\mathbf{p}_{i,j}\}_{j\in[\ell]}\big)$$
$$q_i := \mathsf{CRTRecon}\big(\{\mathbf{m}_j\}_{j\in[\ell]}, \{\mathbf{q}_{i,j}\}_{j\in[\ell]}\big)$$

Next, the simulator runs $\mathsf{CRTSample}(n, \kappa, \{(p_i, q_i)\}_{i\in\mathbf{P}^*})$, and receive as a result either $(\mathtt{success}, p, q)$ or $(\mathtt{failure}, p, q)$ and computes $N = p \cdot q$. For every $i \in \overline{\mathbf{P}^*}$ and $j \in [\ell']$ sample uniformly distributed $\mathbf{N}_{i,j} \leftarrow \mathbb{Z}_{\mathbf{m}_j}$ such that

$$\sum_{i\in[n]} \mathbf{N}_{i,j} \equiv N \pmod{\mathbf{m}_j}$$

The rest of the simulation proceeds as with $\mathcal{S}_{\mathsf{RSAGen}}$, with the exception that there is no need to send $(\mathtt{cheat}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{RSAGen}}$ to learn p and q.

Claim D.4. *The output of $\mathcal{H}_3$ is identically distributed as the output of $\mathcal{H}_2$.*

Proof. The proof follows immediately by the perfect security of additive secret sharing and since biprimes generated by (non-aborting runs of) the protocol are identically distributed according to CRTSample. □

Hybrid $\mathcal{H}_4$. This hybrid experiment follows as $\mathcal{H}_3$ except that instead of running CRTSample to sample p and q, the simulator sends the adversarial shares to $\mathcal{F}_{\mathsf{RSAGen}}$ as $(\mathtt{adv\text{-}input}, \mathsf{sid}, i, p_i, q_i)$ for every $i \in \mathbf{P}^*$ and receives either $(\mathtt{factors}, \mathsf{sid}, p, q)$ or $(\mathtt{biprime}, \mathsf{sid}, N)$ in response. Next, in case the simulator needs p and q for Steps 10 or 12, it sends $(\mathtt{cheat}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{RSAGen}}$ and receives $(\mathtt{factors}, \mathsf{sid}, p, q)$. Finally, the simulator does determine the output for the dummy honest parties, but delivers the output generated by $\mathcal{F}_{\mathsf{RSAGen}}$.

Claim D.5. *The output of $\mathcal{H}_4$ is identically distributed as the output of $\mathcal{H}_3$.*

Proof. The proof follows since the same operations take place in both hybrids, where the only difference is whether CRTSample is run by the simulator or by $\mathcal{F}_{\mathsf{RSAGen}}$. □

Note that $\mathcal{H}_3$ the output of $\mathcal{H}_4$ is identically distributed as $\mathrm{IDEAL}_{\mathcal{F}_{\mathsf{RSAGen}},\mathcal{S}_{\mathsf{RSAGen}},\mathcal{Z}}$. This concludes the proof of Theorem 4.6. □

E Postponed CRTSample Analysis

We give here the postponed proof of the lemma to lower-bound the probability of success for a single invocation of CRTSample, i.e., Lemma 6.1. For this task, it suffices to compute the probability that a randomly chosen value that is coprime to the (κ, n)-Near-primorial Vector $\mathbf{m}$ is truly a prime. We begin by lower-bounding the probability of sampling a true prime, relative to the largest value in $\mathbf{m}$, after which we lower-bound the largest value in $\mathbf{m}$.

Lemma E.1 ([BF01])**.** *Given a (κ, n)-Near-primorial vector $\mathbf{m} \in \mathbb{Z}^\ell$,*

$$\Pr\left[\, p \textit{ is prime} \mid p \leftarrow \mathbb{Z}_{2^\kappa} \;\; \textit{s.t. } p \textit{ is } \mathbf{m}\textit{-coprime}\right] \geq 2.57 \cdot \frac{\ln \mathbf{m}_\ell}{\kappa}$$

Lemma E.2. *If $\mathbf{m} \in \mathbb{Z}^\ell$ is a (κ, n)-Near-primorial vector, then $\mathbf{m}_\ell \in \Omega(\kappa)$*

Proof. Given

$$M := \prod_{i \in [\ell]} \mathbf{m}_i$$

Lemma 3.15 from Rosser and Schoenfeld [RS62] yields

$$\ln(M/2) < \mathbf{m}_\ell \cdot \left(1 + \frac{1}{2 \ln \mathbf{m}_\ell}\right)$$

and per Definition 3.2 we have that M is the largest integer such that $M/2$ is a primorial and $M < 2^{\kappa - \log n}$. From these facts, the prime number theorem gives us $M \in \Omega(2^{\kappa - \log n - \log \kappa})$. Thus we have

$$\mathbf{m}_\ell \cdot \left(1 + \frac{1}{2 \ln \mathbf{m}_\ell}\right) \in \Omega(\kappa - \log n - \log \kappa)$$

and since we know $\mathbf{m}_\ell \geq 3$, we have

$$\mathbf{m}_\ell \in \Omega(\kappa - \log n - \log \kappa)$$

and finally, since Lemma 3.5 constrains $2 \leq n < \kappa$, we have

$$\mathbf{m}_\ell \in \Omega(\kappa)$$

□

Lemma E.3 (Lemma 6.1 restated)**.** *An iteration of* CRTSample *succeeds with probability*

$$\Omega\left(\frac{\log^2 \kappa}{\kappa^2}\right)$$

Proof. Applying Lemma E.1 to the the trial division bound $\Omega(\kappa)$ from Lemma E.2, it follows that the probability of sampling one κ-bit prime is

$$\Omega\left(\frac{\log \kappa}{\kappa}\right)$$

The result follows because p and q are sampled independently. □