Multiparty Generation of an RSA Modulus

Megan Chen
contact.meganchen@gmail.com

Ran Cohen
rancohen@ccs.neu.edu

Jack Doerner
j@ckdoerner.net

Yashvanth Kondi
ykondi@ccs.neu.edu

Eysa Lee
eysa@ccs.neu.edu

Schuyler Rosefield
rosefield.s@northeastern.edu

abhi shelat
abhi@neu.edu

Northeastern University

March 31, 2020

Abstract

We present a new multiparty protocol for the distributed generation of biprime RSA moduli, with security against any subset of maliciously colluding parties assuming oblivious transfer and the hardness of factoring.

Our protocol is highly modular, and its uppermost layer can be viewed as a template that generalizes the structure of prior works and leads to a simpler security proof. We introduce a combined sampling-and-sieving technique that eliminates both the inherent leakage in the approach of Frederiksen et al. (Crypto'18), and the dependence upon additively homomorphic encryption in the approach of Hazay et al. (JCrypt'19). We combine this technique with an efficient, privacy-free check to detect malicious behavior retroactively when a sampled candidate is not a biprime, and thereby overcome covert rejection-sampling attacks and achieve both asymptotic and concrete efficiency improvements over the previous state of the art.

Contents

1 Introduction

A *biprime* is a number N of the form $N = p \cdot q$ where p and q are primes. Such numbers are used as a component of the public key (i.e., the *modulus*) in the RSA cryptosystem [RSA78], with the factorization being a component of the secret key. A long line of research has studied methods for sampling biprimes efficiently; in the early days, the task required specialized hardware and was not considered generally practical [Riv80, Riv84]. In subsequent years, advances in computational power brought RSA into the realm of practicality, and then ubiquity. Given a security parameter κ, the de facto standard method for sampling RSA biprimes involves choosing random κ-bit numbers and subjecting them to the Miller-Rabin primality test [Mil76, Rab80] until two primes are found; these primes are then multiplied to form a 2κ-bit modulus. This method suffices when a single party wishes to generate a modulus, and is permitted to know the associated factorization.

Boneh and Franklin [BF97, BF01] initiated the study of *distributed* RSA modulus generation.[1] This problem involves a set of parties who wish to jointly sample a biprime in such a way that no corrupt and colluding subset (below some defined threshold size) can learn the biprime's factorization.

It is clear that applying generic multiparty computation (MPC) techniques to the standard sampling algorithm yields an impractical solution: implementing the Miller-Rabin primality test requires repeatedly computing $a^{p-1} \pmod{p}$, where p is (in this case) secret, and so such an approach would require the generic protocol to evaluate a circuit containing many modular exponentiations over κ bits each. Instead, Boneh and Franklin [BF97, BF01] constructed a new biprimality test that generalizes Miller-Rabin and avoids computing modular exponentiations with secret moduli. Their test carries out all exponentiations modulo the public biprime N itself, and this allows the exponentiations to be carried out locally by the parties. Furthermore, they introduced a three-phase structure for the overall sampling protocol, which all subsequent works have 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 them, culling candidates with small factors.

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

3. **Biprimality Testing**: using a distributed protocol, N is tested for biprimality. If N is not a biprime, then the process is repeated.

[1] Prior works generally consider RSA *key generation* and include steps for generating shares of e and d such that $e \cdot d \equiv 1 \pmod{\varphi(N)}$. This work focuses only on the task of sampling the RSA modulus N. Prior techniques can be applied to sample (e, d) after sampling N, and the distributed generation of an RSA modulus has standalone applications, such as for generating the trusted setup required by verifiable delay functions [Pie19, Wes19]; consequently, we omit further discussion of e and d.

The seminal work of Boneh and Franklin considered the semi-honest n-party setting with an honest majority of participants. Many extensions and improvements followed (as detailed in Section 1.3), the most notable of which (for our purposes) are two recent works that achieve malicious security against a dishonest majority. In the first, Hazay et al. [HMRT12, HMR$^+$19] proposed an n-party protocol in which both sieving and modulus reconstruction are achieved via additively homomorphic encryption. Specifically, they rely upon both ElGamal and Paillier encryption, and in order to achieve malicious security, they use zero-knowledge proofs for a variety of relations over the ciphertexts. Thus, their protocol represents a substantial advancement in terms of its security guarantee, but this comes at the cost of additional complexity assumptions and an intricate proof, and also at substantial concrete cost, due to the use of many custom zero-knowledge proofs.

The subsequent protocol of Frederiksen et al. [FLOP18] (the second recent work of note) relies mainly on oblivious transfer (OT), which they use to perform both sieving and, via Gilboa's classic multiplication protocol [Gil99], modulus reconstruction. They achieved malicious security using the folklore technique in which a "Proof of Honesty" is evaluated as the last step and demonstrated practicality by implementing their protocol; however, it is not clear how to extend their approach to more than two parties in a straightforward way. Moreover, their approach to sieving admits selective-failure attacks, for which they account by including some leakage in the functionality. It also permits a malicious adversary to selectively and *covertly* induce false negatives (i.e., force the rejection of true biprimes after the sieving stage), a property that is again modeled in their functionality. In conjunction, these attributes degrade *security*, because the adversary can rejection-sample biprimes based on the additional leaked information, and *efficiency*, because ruling out malicious false-negatives involves running sufficiently many instances to make the probability of statistical failure in all instances negligible.

Thus, given the current state of the art, it remains unclear whether one can sample an RSA modulus among two parties (one being malicious) without leaking additional information or permitting covert rejection sampling, or whether one can sample an RSA modulus among many parties (all but one being malicious) without involving heavy cryptographic primitives such as additively homomorphic encryption, and their associated performance penalties. In this work, we present a protocol which efficiently achieves both tasks.

1.1 Results and Contributions

A Clean Functionality. We define $\mathcal{F}_{\mathsf{RSAGen}}$, a simple, natural functionality for sampling biprimes from the same well-known distribution used by prior works [BF01, HMR$^+$19, FLOP18], with no leakage or conflation of sampling failures with adversarial behavior.

A Modular Protocol, with Natural Assumptions. We present a protocol π_{RSAGen} in the $(\mathcal{F}_{\mathsf{AugMul}}, \mathcal{F}_{\mathsf{Biprime}})$-hybrid model, where $\mathcal{F}_{\mathsf{AugMul}}$ is an augmented multiplier functionality and $\mathcal{F}_{\mathsf{Biprime}}$ is a biprimality-testing functionality, and prove that it UC-realizes $\mathcal{F}_{\mathsf{RSAGen}}$ in the malicious setting, assuming the hardness of factoring. More specifically, we prove:

Theorem 1.1 (Main Security Theorem, Informal)**.** *In the presence of a PPT malicious adversary corrupting any subset of parties, $\mathcal{F}_{\mathsf{RSAGen}}$ can be securely computed with abort in the $(\mathcal{F}_{\mathsf{AugMul}}, \mathcal{F}_{\mathsf{Biprime}})$-hybrid model, assuming the hardness of factoring.*

Additionally, because our security proof relies upon the hardness of factoring only when the adversary cheats, we find to our surprise that our protocol achieves *perfect* security against semi-honest adversaries.

Theorem 1.2 (Semi-Honest Security Theorem, Informal)**.** *In the presence of a computationally unbounded semi-honest adversary corrupting any subset of parties, $\mathcal{F}_{\mathsf{RSAGen}}$ can be computed with perfect security in the $(\mathcal{F}_{\mathsf{AugMul}}, \mathcal{F}_{\mathsf{Biprime}})$-hybrid model.*

Supporting Functionalities and Protocols. We define $\mathcal{F}_{\mathsf{Biprime}}$, a simple, natural functionality for biprimality testing, and show that it is UC-realized in the semi-honest setting by a well known protocol of Boneh and Franklin [BF01], and in the malicious setting by a derivative of the protocol of Frederiksen et al. [FLOP18]. We believe this dramatically simplifies the composition of these two protocols, and as a consequence, leads to a simpler analysis. Either protocol can be based exclusively upon oblivious transfer.

We also define $\mathcal{F}_{\mathsf{AugMul}}$, a functionality for sampling and multiplying secret-shared values in a special form derived from the Chinese Remainder Theorem. In the context of π_{RSAGen}, this functionality allows us to efficiently sample numbers in a specific range, with no small factors, and then compute their product. We prove that it can be UC-realized exclusively from oblivious transfer, using derivatives of well known multiplication protocols [DKLS18, DKLS19].

Asymptotic Efficiency. We perform an asymptotic analysis of our composed protocols and find that our semi-honest protocol is a factor of $\kappa/\log\kappa$ more bandwidth-efficient than that of Frederiksen et al. [FLOP18]. Our maliciously secure protocol is a factor of κ/s more efficient than theirs in the optimistic case (when parties follow the protocol), and a factor of κ more efficient when parties deviate from the protocol. Recall that κ is the bit-length of the primes p and q, and s is a statistical security parameter. Frederiksen et al. claim in turn that their protocol is strictly superior to the protocol of Hazay et al. [HMR^{+}19] with respect to asymptotic bandwidth performance.

Concrete Efficiency. We perform a closed-form concrete analysis of our protocol (with some optimizations, including the use of random oracles), and find that it outperforms the protocol of Frederiksen et al. (the most efficient prior work) by a wide margin in terms of communication.

1.2 Overview of Techniques

Constructive Sampling and Efficient Modulus Reconstruction. Most prior works use rejection sampling to generate a pair of candidate primes, and then multiply those primes together in a separate step. Specifically, they sample a shared value $p \leftarrow [0, 2^\kappa)$ uniformly, and then perform a trial-division protocol repeatedly, discarding both the value and the work that has gone into testing it if one round of trial division fails. This represents a substantial amount of wasted work in expectation. Furthermore, Frederiksen et al. [FLOP18] report that multiplication of candidates after sieving accounts for two thirds of their concrete cost.

We propose a different approach that leverages the Chinese Remainder Theorem (CRT) to *constructively* sample a pair of candidate primes and multiply them together efficiently. A similar sieving approach (in spirit) was initially formulated as an optimization in a different setting by Malkin et al. [MWB99]. The CRT implies an isomorphism between a set of values, each in a field modulo a distinct prime, and a single value in a ring modulo the product of those primes (i.e., $\mathbb{Z}_{m_1} \times \ldots \times \mathbb{Z}_{m_\ell} \simeq \mathbb{Z}_{m_1 \cdot \ldots \cdot m_\ell}$). We refer to the set of values as the *CRT form* or *CRT representation* of the single value to which they are isomorphic. We formulate a sampling mechanism based on this isomorphism as follows: for each of the first $O(\kappa/\log\kappa)$ odd primes, the parties jointly (and efficiently) sample shares of a value that is nonzero modulo that prime. These values are the shared CRT form of a single κ-bit value that is guaranteed to be indivisible by any prime in the set sampled against. For technical reasons, we sample two such candidates simultaneously.

Rather than converting pairs of candidate primes from CRT form to standard form, and then multiplying them, we instead multiply them component-wise in CRT form, and then convert the product to standard form to complete the protocol. This effectively replaces a single "full-width" multiplication of size κ with $O(\kappa/\log\kappa)$ individual multiplications, each of size $O(\log\kappa)$. We intend to perform multiplication via an OT-based protocol, and the computation and communication complexity of such protocols grows at least with the square of their input length, even in the semi-honest case [Gil99]. Thus in the semi-honest case, our approach yields an overall complexity of $O(\kappa\log\kappa)$, as compared to $O(\kappa^2)$ for a single full-width multiplication. In the malicious case, combining the best known multiplier construction [DKLS18, DKLS19] with the most efficient known OT extension scheme [BCG$^+$19] yields a complexity that also grows with the product of the input length and a statistical parameter s, and so our approach achieves an overall complexity of $O(\kappa\log\kappa + \kappa\cdot s)$, as compared to $O(\kappa^2 + \kappa\cdot s)$ for a single full-width malicious multiplication. Via closed-form analysis, we show that this asymptotic improvement is also reflected concretely.

Achieving Security with Abort Efficiently. The fact that we sample primes in CRT form also plays a crucial role in our security analysis. Unlike the work of Frederiksen et al. [FLOP18], our protocol achieves the *standard*, intuitive notion of security with abort: the adversary can instruct the functionality to abort regardless of whether a biprime is successfully sampled, and the honest parties are always made aware of such adversarial aborts. There is, in other words, absolutely no conflation of sampling failures with adversarial behavior. For the sake of efficiency, our protocol permits the adversary to cheat prior to biprimality testing, and then rules out such cheats retroactively using one of two strategies. In the case that a biprime is successfully sampled, adversarial behavior is ruled out retroactively in a privacy-preserving fashion using well-known but moderately expensive techniques, which is tolerable only because it need not be done more than once. In the case that a sampled value is not a biprime, however, the inputs to the sampling protocol are revealed to all parties, and the retroactive check is carried out in the clear. Proving the latter approach secure turns out to be surprisingly subtle.

The challenge arises from the fact that the simulator must simulate the protocol transcript for the OT-multipliers on behalf of the honest parties without knowing their inputs. Later, if the sampling-protocol inputs are revealed, the simulator must "explain" how the simulated transcript is consistent with the true inputs of the honest parties. Specifically, in maliciously secure OT-multipliers of the sort we use [DKLS18, DKLS19], the OT receiver (Bob) uses a high-entropy encoding of his input, and the sender (Alice) can, by cheating, learn a one-bit predicate of this encoding. Before Bob's true input is known to the simulator, it must pick an encoding at random. When Bob's input is revealed, the simulator must find an encoding of his input which is consistent with the predicate on the random encoding that Alice has learned. This task closely resembles solving a random instance of subset sum.

We are able to overcome this difficulty because our multiplications are performed component-wise over CRT-form representations of their operands. Because each component is of size $O(\log \kappa)$ bits, the simulator can simply guess random encodings until it finds one that matches the required constraints. We show that this strategy succeeds in strict polynomial time, and that it induces a distribution statistically close to that of the real execution.

This form of "privacy-free" malicious security (wherein honest behavior is verified at the cost of sacrificing privacy) leads to considerable efficiency gains in our case: it is up to a multiplicative factor of s (the statistical parameter) cheaper than the privacy-preserving check used in the case that a candidate passes the biprimality test (and the one used in prior OT-multipliers [DKLS18, DKLS19]). Since most candidates fail the biprimality test, using the privacy-free check to verify that they were generated honestly results in substantial savings.

Biprimality Testing as a Black Box. We specify a functionality for biprimality testing, and prove that it can be realized by a maliciously secure version of the Boneh-Franklin biprimality test. Our functionality has a clean interface and does not, for example, require its inputs to be authenticated to ensure that they were actually generated by the sampling phase of the protocol. The key insight that allows us to achieve this level of modularity is a reduction to factoring: if an adversary is able to cheat by supplying incorrect inputs to the biprimality test, relative to a candidate biprime N, and the biprimality test succeeds, then we show that the adversary can be used to factor biprimes. We are careful to rely on this reduction only in the case that N is actually a biprime, and to prevent the adversary from influencing the distribution of candidates.

The Benefits of Modularity. We claim as a contribution the fact that modularity has yielded both a simple protocol description and a reasonably simple proof of security. We believe that this approach will lead to derivatives of our work with stronger security properties or with security against stronger adversaries. As a first example, we prove that a semi-honest version of our protocol (differing only in that it omits the retroactive consistency check in the protocol's final step) achieves *perfect* security. We furthermore observe that in the malicious setting, instantiating $\mathcal{F}_{\mathsf{Biprime}}$ and $\mathcal{F}_{\mathsf{AugMul}}$ with security against adaptive adversaries yields an RSA modulus sampling protocol that is adaptively secure.

Similarly, only minor adjustments to the main protocol are required to achieve security with *identifiable abort* [IOZ14, CL17]. If we assume that the underlying functionalities $\mathcal{F}_{\mathsf{AugMul}}$ and $\mathcal{F}_{\mathsf{Biprime}}$ are instantiated with identifiable abort, then it remains only to ensure the use of consistent inputs across these functionalities, and to detect which party has provided inconsistent inputs if an abort occurs. This can be accomplished by augmenting $\mathcal{F}_{\mathsf{Biprime}}$ with an additional interface for revealing the input values provided by all the parties upon global request (e.g., when the candidate N is not a biprime). A technical proof of this extension is beyond the scope of this work; we focus instead on the advancements our framework yields in the setting of security with abort.

1.3 Additional Related Work

Frankel, MacKenzie, and Yung [FMY98] adjusted the protocol of Boneh and Franklin [BF97] to achieve security against malicious adversaries in the honest-majority setting. Their main contribution was the introduction of a method for robust distributed multiplication over the integers. Cocks [Coc97] proposed a method for multiparty RSA key generation under heuristic assumptions, and later attacks by Coppersmith (see [Coc98]) and Joye and Pinch [JP99] suggest this method may be insecure. Poupard and Stern [PS98] presented a malicious-secure two-party protocol based on oblivious transfer. Gilboa [Gil99] achieved improved efficiency in the semi-honest two-party model, and introduced a novel method for multiplication from oblivious transfer, from which our own multipliers ultimately derive.

Malkin, Wu, and Boneh [MWB99] implemented the protocol of Boneh and Franklin and introduced an optimized sieving method similar in spirit to ours. In particular, their protocol generates sharings of random values in $\mathbb{Z}_M^*$ (where M is a primorial modulus) during the sieving phase, instead of naïve random candidates for primes p and q. However, their method produces *multiplicative* sharings of p and q, which are converted into additive sharings for biprimality testing via a semi-honest honest-majority protocol. This conversion requires

Protocol	Parties	Corruptions	Security	Channels	Assumptions
[BF97]	$n \geq 3$	$t < n/2$	Semi-honest	Priv	None
[FMY98]	$n \geq 3$	$t < n/2$	Malicious	Priv, BC	DL
[PS98]	$n = 2$	$t = 1$	Malicious	Auth	OT
[Gil99]	$n = 2$	$t = 1$	Semi-honest	Auth	OT
[ACS02]	$n \geq 3$	$t < n/2$	Semi-honest	Priv	None
[DM10]	$n = 3$	$t = 1$	Malicious	Priv, BC	CRS, SRSA
[HMR+19]	$n \geq 2$	$t < n$	Malicious	Auth, BC	DCR, DDH
[FLOP18]	$n = 2$	$t = 1$	Malicious	Auth	OT
This Work	$n \geq 2$	$t < n$	Malicious	Auth, BC	OT, Factoring

Table 1.3: Comparison of Prior Works. *Priv*, *Auth*, and *BC* stand for private, authenticated, and broadcast channels, respectively. *DL* stands for discrete log, *OT* for oblivious transfer, *CRS* for a common reference string, *SRSA* for Strong RSA, *DCR* for decisional composite residuosity, and *DDH* for decisional Diffie-Hellman.

rounds linear in the party count, and it is unclear how to adapt it to tolerate a malicious majority of parties without a significant performance penalty.

Algesheimer, Camenish, and Shoup [ACS02] described a method to compute a distributed version of the Miller-Rabin test: they use secret-sharing conversion techniques reliant on approximations of $1/p$ to compute exponentiations modulo a shared p. However, each invocation of their Miller-Rabin test still has complexity in $O(\kappa^3)$ per party, and their overall protocol has complexity in $O(\kappa^5/\log^2\kappa)$, with $\Theta(\kappa)$ rounds of interaction. Concretely, Damgård and Mikkelsen [DM10] estimate that 10000 rounds are required to sample a 2000-bit biprime using this method. Damgård and Mikkelsen also extended their work to improve both its complexity and round count by several orders of magnitude, and to achieve malicious security in the honest-majority setting. Their protocol is at least a factor of $O(\kappa)$ better than that of Algesheimer, Camenish, and Shoup, but it still requires *hundreds* of rounds. We were not able to compute an explicit complexity analysis of their approach. We give a summary of prior works in Table 1.3, for ease of comparison.

In a follow-up work, Chen et al. [CHI+20] adapt our CRT-based sampling mechanism to work with multipliers and zero-knowledge proofs based on additively homomorphic encryption. They focus on the setting wherein there is a powerful, untrusted aggregator, and many weak clients. Via implementation, they show that this approach is sufficiently efficient for real-world use, even with thousands of participants spread around the world.

1.4 Organization

Basic notation and background information are given in Section 2. Our ideal biprime-sampling functionality is defined in Section 3, and we give a protocol that realizes it in Section 4. In Section 5, we present our biprimality-testing protocol. In Section 6 we give an efficiency analysis. We defer the details of our multiplication protocol to the appendices.

2 Preliminaries

Notation. 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 $\mathsf{sans\text{-}serif}$ font, and string literals are set in $\mathtt{slab\text{-}serif}$ font. We use mod to indicate the modulus operator, while $\pmod{m}$ at the end of a line indicates 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. We use s to represent the statistical parameter. Where concrete efficiency is concerned, we introduce a second computational security parameter, λ, which represents the length of a symmetric key of equivalent strength to a biprime of length 2κ.[2] κ and λ must vary together, and a recommendation for the relationship between them has been laid down by NIST [Bar16].

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, we place a $*$ in the 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 $\mathbf{y}$.

Universal Composability. We prove our protocols secure in the Universal Composability (UC) framework, and use standard UC notation. In Appendix A, we give a high-level overview and refer the reader to [Can01] for 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 inappropriately, send messages out of order, etc. For convenience, we provide a function GenSID, which takes *any* number of arguments and deterministically derives a unique Session ID from those arguments.

[2] In other words, a biprime of length 2κ provides λ bits of security.

Chinese Remainder Theorem. The Chinese Remainder Theorem (CRT) defines an isomorphism between a set of residues modulo a set of respective coprime values and a single value modulo the product of the same set of coprime values. This forms the basis of our sampling procedure.

Theorem 2.1 (CRT)**.** *Let $\mathbf{m}$ be a vector of coprime positive integers and let $\mathbf{x}$ be a vector of numbers such that $|\mathbf{m}| = |\mathbf{x}| = \ell$ and $0 \leq \mathbf{x}_j < \mathbf{m}_j$ for all $j \in [\ell]$, and finally let $M := \prod_{j\in[\ell]} \mathbf{m}_j$. Under these conditions there exists a unique value y such that $0 \leq y < M$ and $y \equiv \mathbf{x}_j \pmod{\mathbf{m}_j}$ for every $j \in [\ell]$.*

We refer to $\mathbf{x}$ as the *CRT form* of y with respect to $\mathbf{m}$. For completeness, we give the CRTRecon algorithm, which finds the unique y given $\mathbf{m}$ and $\mathbf{x}$.

Algorithm 2.2. $\mathsf{CRTRecon}(\mathbf{m}, \mathbf{x})$

1. With $\ell := |\mathbf{m}|$, 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 Functionality

We begin this section by discussing the distribution of biprimes from which we sample, and thus the precise factoring assumption that we make, and then we give an efficient sampling algorithm and an ideal functionality that computes it.

3.1 Factoring Assumptions

The standard factoring experiment (Experiment 3.1) as formalized by Katz and Lindell [KL15] is parametrized by an adversary $\mathcal{A}$ and a biprime-sampling algorithm GenModulus. On input 1^κ, this algorithm returns (N, p, q), where $N = p \cdot q$, and p and q are κ-bit primes.[3]

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

[3] Technically, Katz and Lindell specify that sampling failures are permitted with negligible probability, and require GenModulus to run in strict polynomial time. We elide this detail.

factoring assumption with respect to this common $\mathsf{GenModulus}$ function states that for every PPT adversary $\mathcal{A}$ there exists a negligible function negl such that

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

Because *efficiently* sampling according to this uniform biprime 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 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 $[0, 2^{\kappa - \log n})$,[4] 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 $q \equiv 1 \pmod p$.[5]

Algorithm 3.2. $\mathsf{BFGM}(n, \kappa)$

1. For $i \in [n]$, sample $p_i \leftarrow \left[0, 2^{\kappa - \log n}\right)$ and $q_i \leftarrow \left[0, 2^{\kappa - \log n}\right)$ 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.

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 $\mathcal{A}$,

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

3.2 The Distributed Biprime-Sampling Functionality

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

[4] Boneh and Franklin [BF01] are somewhat ambiguous as to whether the lower bound on each share is $2^{\kappa - \log n - 1}$ or 0. We take the latter interpretation, as have prior works [HMR+19, FLOP18]. We do not believe the difference to be important.

[5] 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.[6] Furthermore, we give CRTSample a specific distribution of failures that 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 begin with a few useful notions.

Definition 3.3 (Primorial Number)**.** *The i^{th}* primorial number *is defined to be the product of the first i prime numbers.*

Definition 3.4 ((κ, n)-Near-Primorial Vector)**.** *Let ℓ be the largest number such that the ℓ^{th} primorial number is less than $2^{\kappa-\log n-1}$, and let $\mathbf{m}$ be a vector of length ℓ such that $\mathbf{m}_1 = 4$ and $\mathbf{m}_2, \ldots, \mathbf{m}_\ell$ are the odd factors of the ℓ^{th} primorial number, in ascending order. $\mathbf{m}$ is the unique* (κ, n)-near-primorial vector.

Definition 3.5 ($\mathbf{m}$-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 3.6. CRTSample$(n, \kappa, \{(p_i, q_i)\}_{i\in\mathbf{P}^*})$

1. Let $\mathbf{m}$ be the (κ, n)-near-primorial vector, with length ℓ, and let M be the product of $\mathbf{m}$.

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

$$p := \sum_{i\in[n]} p_i \qquad \text{and} \qquad q := \sum_{i\in[n]} q_i$$

   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 knowledge of $n-1$ integer shares of the factors p and q does not give the adversary 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, wherein the adversary is allowed to choose its own shares.

[6] CRTSample never outputs biprimes with factors smaller than κ, whereas BFGM outputs such biprimes with negligible probability. The discrepancy of share ranges can be remedied by using non-integer values of κ with BFGM.

Lemma 3.7 ([BF01, HMR+19]). *Let $n < \kappa$ and let $(\mathcal{A}_1, \mathcal{A}_2)$ be a pair of PPT algorithms. For $(\mathsf{state}, \{(p_i, q_i)\}_{i\in[n-1]}) \leftarrow \mathcal{A}_1(1^n, 1^\kappa)$, let N be a biprime sampled by running $\mathsf{CRTSample}(n, \kappa, \{(p_i, q_i)\}_{i\in[n-1]})$. If $\mathcal{A}_2(\mathsf{state}, N)$ 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 $\mathsf{Factor}_{\mathcal{B},\mathsf{BFGM}}(n, \kappa)$.*

Multiparty functionality. Our ideal functionality $\mathcal{F}_{\mathsf{RSAGen}}$ is a natural embedding of 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 CRTSample with these inputs when invoked. It 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 CRTSample per invocation of $\mathcal{F}_{\mathsf{RSAGen}}$ enables significant freedom in the use of $\mathcal{F}_{\mathsf{RSAGen}}$, because it can be composed in different ways to tune the trade-off between resource usage and execution time. It also simplifies the analysis of the protocol π_{RSAGen} that realizes $\mathcal{F}_{\mathsf{RSAGen}}$, because the analysis is made independent of the success rate of the sampling procedure.

The functionality may not deliver N to the honest parties for one of two reasons: either CRTSample failed to sample a biprime, or the adversary caused the computation to abort. In either case, 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 CRTSample algorithm. In both cases, the p and q output by 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 3.8. $\mathcal{F}_{\mathsf{RSAGen}}(\kappa, n)$**. Distributed Biprime Sampling**

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

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 output, as appropriate. Terminate successfully.
- If $\texttt{cheat}$ 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 Biprime-Sampling Protocol

In this section, we present the distributed biprime-sampling protocol π_{RSAGen}, with which we realize $\mathcal{F}_{\mathsf{RSAGen}}$. We begin with a high-level overview, and then in Section 4.2, we formally define the two ideal functionalities on which our protocol relies, after which in Section 4.3 we give the protocol itself. In Section 4.4, we present proof sketches of semi-honest and malicious security.

4.1 High-Level Overview

As described in the Introduction, our protocol derives from that of Boneh and Franklin [BF01], the main technical differences relative to other recent Boneh-Franklin derivatives [HMR+19, FLOP18] being the modularity with which it is described and proven, and the use of CRT-based sampling. Our protocol has three main phases, which we now describe in sequence.

Candidate Sieving. In the first phase of our protocol, the parties jointly sample two κ-bit candidate primes p and q without any small factors, and multiply them to learn their product N. Our protocol achieves these tasks in a unified, integrated way, thanks to the Chinese Remainder Theorem.

Consider a prime m and a set of shares x_i for $i \in [n]$ over the field $\mathbb{Z}_m$. As in the description of $\mathsf{CRTRecon}$, let a and b be defined such that $a \cdot b \equiv 1 \pmod{m}$, and let M be an integer. Observe that if m divides M, then

$$\sum_{i\in[n]} x_i \not\equiv 0 \pmod{m} \quad\Longrightarrow\quad \sum_{i\in[n]} a \cdot b \cdot x_i \bmod M \not\equiv 0 \pmod{m} \tag{1}$$

Now consider a vector of coprime integers $\mathbf{m}$ of length ℓ, and let M be their product. Let $\mathbf{x}$ be a vector, each element secret shared over the fields defined by the corresponding element of $\mathbf{m}$, and let $\mathbf{a}$ and $\mathbf{b}$ be defined as in $\mathsf{CRTRecon}$ (i.e., $\mathbf{a}_j := M/\mathbf{m}_j$ and $\mathbf{a}_j \cdot \mathbf{b}_j \equiv 1 \pmod{\mathbf{m}_j}$). We can see that for any $k, j \in [\ell]$ such that $k \neq j$,

$$\mathbf{a}_j \equiv 0 \pmod{\mathbf{m}_k} \quad\Longrightarrow\quad \sum_{i\in[n]} \mathbf{a}_j \cdot \mathbf{b}_j \cdot \mathbf{x}_{i,j} \bmod M \equiv 0 \pmod{\mathbf{m}_k} \tag{2}$$

and the conjunction of Equations 1 and 2 gives us

$$\sum_{j\in[\ell]}\sum_{i\in[n]}\mathbf{a}_j\cdot\mathbf{b}_j\cdot\mathbf{x}_{i,j} \bmod M \equiv \sum_{i\in[n]}\mathbf{x}_{i,k} \pmod{\mathbf{m}_k}$$

for all $k\in[\ell]$. Observe that this holds regardless of which order we perform the sums in, and regardless of whether the mod M operation is done at the end, or between the two sums, or not at all.

It follows then that we can sample n shares for an additive secret sharing over the integers of a κ-bit value x (distributed between 0 and $n\cdot M$) by choosing $\mathbf{m}$ to be the (κ,n)-near-primorial vector (per Definition 3.4), instructing each party $\mathcal{P}_i$ for $i\in[n]$ to pick $\mathbf{x}_{i,j}$ locally for $j\in[\ell]$ such that $0\le\mathbf{x}_{i,j}<\mathbf{m}_j$, and then instructing each party to *locally* reconstruct $x_i := \mathsf{CRTRecon}(\mathbf{m},\mathbf{x}_{i,*})$, its share of x. It furthermore follows that if the parties can contrive to ensure that

$$\sum_{i\in[n]}\mathbf{x}_{i,j}\not\equiv 0 \pmod{\mathbf{m}_j} \tag{3}$$

for $j\in[\ell]$, then x will not be divisible by any prime in $\mathbf{m}$.

Observe next that if the parties sample two shared vectors $\mathbf{p}$ and $\mathbf{q}$ as above (corresponding to the candidate primes p and q) and compute a shared vector $\mathbf{N}$ of identical dimension such that

$$\sum_{i\in[n]}\mathbf{p}_{i,j}\cdot\sum_{i\in[n]}\mathbf{q}_{i,j}\equiv\sum_{i\in[n]}\mathbf{N}_{i,j} \pmod{\mathbf{m}_j} \tag{4}$$

for all $j\in[\ell]$, then it follows that

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

and from this it follows that the parties can calculate integer shares of $N=p\cdot q$ by multiplying $\mathbf{p}$ and $\mathbf{q}$ together element-wise using a modular-multiplication protocol for linear secret shares, and then locally running $\mathsf{CRTRecon}$ on the output to reconstruct N. In fact, our sampling protocol makes use of a special functionality $\mathcal{F}_{\mathsf{AugMul}}$, which samples $\mathbf{p}$, $\mathbf{q}$, and $\mathbf{N}$ simultaneously such that the conditions in Equations 3 and 4 hold.

There remains one problem: our vector $\mathbf{m}$ was chosen for sampling integer-shared values between 0 and $n\cdot M$ (with each share no larger than M), but N might be as large as $n^2\cdot M^2$. In order to avoid wrapping during reconstruction of N, we must reconstruct with respect to a *larger* vector of primes (while continuing to sample with respect to a smaller one). Let $\mathbf{m}$ now be of length ℓ', and let ℓ continue to denote the length of the prefix of $\mathbf{m}$ with respect to which sampling is performed. After sampling the initial vectors $\mathbf{p}$, $\mathbf{q}$, and $\mathbf{N}$, each party $\mathcal{P}_i$ for $i\in[n]$ must extend $\mathbf{p}_{i,*}$ locally to ℓ' elements, by computing

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

for $j \in [\ell + 1, \ell']$, and then likewise for $\mathbf{q}_{i,*}$. Finally, the parties must use a modular-multiplication protocol to compute the appropriate extension of $\mathbf{N}$; from this extended $\mathbf{N}$, they can reconstruct shares of $N = p \cdot q$. They swap these shares, and thus each party ends the Sieving phase of our protocol with a candidate biprime N and an integer share of each of its factors, p_i and q_i.

Each party completes the first phase by performing a local trial division to check if N is divisible by any prime smaller than some bound B (which is a parameter of the protocol). The purpose of this step is to reduce the number of calls to $\mathcal{F}_{\mathsf{Biprime}}$ and thus improve efficiency.

Biprimality Test. The parties jointly execute a biprimality test, where every party inputs the candidate N and its shares p_i and q_i, and receives back a biprimality indicator. This phase essentially comprises a single call to a functionality $\mathcal{F}_{\mathsf{Biprime}}$, which allows an adversary to force spurious negative results, but never returns false positive results. Though this phase is simple, much of the subtlety of our proof concentrates here: we show via a reduction to factoring that cheating parties have a negligible chance to pass the biprimality test if they provide wrong inputs. This eliminates the need to authenticate the inputs in any way.

Consistency Check. To achieve malicious security, the parties must ensure that none among them cheated during the previous stages in a way that might influence the result of the computation. This is what we have previously termed the retroactive consistency check. If the biprimality test indicated that N is *not* a biprime, then the parties use a special interface of $\mathcal{F}_{\mathsf{AugMul}}$ to reveal the shares they used during the protocol, and then they verify locally and independently that p and q are not both primes. If the biprimality test indicated that N is a biprime, then the parties run a secure test (again via a special interface of $\mathcal{F}_{\mathsf{AugMul}}$) to ensure that length extensions of $\mathbf{p}$ and $\mathbf{q}$ were performed honestly. To achieve semi-honest security, this phase is unnecessary, and the protocol can end with the biprimality test.

4.2 Ideal Functionalities Used in the Protocol

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. It is meant to help in manipulating CRT-form secret shares. It contains five basic interfaces.

- The `sample` interface 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) from every corrupted party $\mathcal{P}_i$, and then samples a triplet $(x_j, y_j, z_j) \leftarrow \mathbb{Z}_m^3$ for every honest $\mathcal{P}_j$ conditioned on

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

  In the context of π_{RSAGen}, this is used to sample CRT-shares of p and q.

- The `input` and `multiply` interfaces, taken together, allow the parties to load shares (with respect to some small prime modulus m) into the functionality's memory, and later perform modular multiplication on two sets of shares that are associated with the same modulus. That is, given a prime m, each party $\mathcal{P}_i$ inputs x_i and, independently, y_i, and when the parties request a product, with each corrupt party $\mathcal{P}_j$ also supplying its own an output share z_j, the functionality samples a share of z from $\mathbb{Z}_m$ for each honest party subject to

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

  In the context of π_{RSAGen}, this interface is used to perform length-extension on CRT-shares of p and q.

- The `check` interface allows the parties to securely compute a predicate over the set of stored values. In the context of π_{RSAGen}, this is used to check that the CRT-share extension of p and q has been performed correctly, when N is a biprime.

- The `open` interface allows the parties to retroactively reveal their inputs to one another. In the context of π_{RSAGen}, this is used to verify the sampling procedure and biprimality test when N is not a biprime.

These five interfaces suffice for the malicious version of the protocol, and the first three alone suffice for the semi-honest version. We make a final adjustment, which leads to a substantial efficiency improvement in the protocol with which we realize $\mathcal{F}_{\mathsf{AugMul}}$ (which we describe in Appendix B). Specifically, we give the adversary an interface by which it can request that any stored value be leaked to itself, and by which it can (arbitrarily) determine the output of any call to the `sample` or `multiply` interfaces. However, if the adversary uses this interface, the functionality remembers, and informs the honest parties by aborting when the `check` or `open` interfaces is used.

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

This functionality is parametrized 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 $\overline{\mathbf{P}^*} := [n]\backslash\mathbf{P}^*$.

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 $(\texttt{sampled-product}, \mathsf{sid}_1, \mathsf{sid}_2, x_i, y_i, z_i)$ 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

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

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.

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 a candidate biprime N, along with shares of its factors p and q. It checks whether p and q are primes and whether $N = p \cdot q$. The adversary is given an additional interface, by which it can ask the functionality to leak the honest parties' inputs, but when this interface is used then the functionality reports to the honest parties that N is not a biprime, even if it is one.

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

This functionality is parametrized by the integer M and the party-count 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 $0 \leq q < M$, then send a message $(\texttt{biprime}, \mathsf{sid})$ to $\mathcal{S}$. If $\mathcal{S}$ responds with $(\texttt{proceed}, \mathsf{sid})$, then output $(\texttt{biprime}, \mathsf{sid})$ to all parties as adversarially delayed output. If $\mathcal{S}$ responds with $(\texttt{cheat}, \mathsf{sid})$[a], or if any of the previous predicates is false, then 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.

[a] Semi-honest adversaries are forbidden to send the cheat instruction.

Realizations In Appendix B, we discuss a protocol to realize $\mathcal{F}_{\mathsf{AugMul}}$, and in Section 5, we propose a protocol to realize $\mathcal{F}_{\mathsf{Biprime}}$. Both make use of generic MPC, but in such a way that no generic MPC is required unless N is a biprime.

4.3 The Protocol Itself

We refer the reader back to Section 4.1 for an overview of our protocol. We have mentioned that it requires a vector of coprime values, which is prefixed by the (n, κ)-near-primorial vector. We now give this vector a precise definition. Note that the efficiency of our protocol relies upon this vector, because we use its contents to sieve candidate primes. Since smaller numbers are more likely to be factors for the candidate primes, we choose the largest allowable set of the smallest sequential primes.

Definition 4.3 ((n, κ)-Compatible Parameter Set)**.** *Let ℓ' be the smallest number such that the ${\ell'}^{th}$ primorial number is greater than $2^{2\kappa-1}$, and let $\mathbf{m}$ be a vector of length ℓ' such that $\mathbf{m}_1 = 4$ and $\mathbf{m}_2, \ldots, \mathbf{m}_{\ell'}$ are the odd factors of the ${\ell'}^{th}$ primorial number, in ascending order. $(\mathbf{m}, \ell', \ell, M)$ is the (κ, n)-*compatible parameter set *if $\ell < \ell'$ and the prefix of $\mathbf{m}$ of length ℓ is the (κ, n)-near-primorial vector per Definition 3.4, and if M is the product of this prefix.*

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

This protocol is parametrized by the RSA prime length κ, the number of parties n, and the trial-division bound B. Let $(\mathbf{m}, \ell', \ell, M)$ be the (κ, n)-compatible parameter set, per Definition 4.3. In this protocol the parties have access to the functionalities $\mathcal{F}_{\mathsf{AugMul}}(n)$ and $\mathcal{F}_{\mathsf{Biprime}}(n, M)$.

Candidate Sieving:

1. Upon receiving input $(\texttt{sample}, \mathsf{sid})$ from the environment, the parties begin the protocol. Every party $\mathcal{P}_i$ for $i \in [n]$ computes three vectors of session IDs

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

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

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 $j \in [2, \ell']$, 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

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

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 compute

$$p_{i'} := \mathsf{CRTRecon}\left(\mathbf{m}, \mathbf{p}_{i',*}\right) \qquad \text{and} \qquad q_{i'} := \mathsf{CRTRecon}\left(\mathbf{m}, \mathbf{q}_{i',*}\right)$$

for all $i' \in [n]$ and to return 1 if and only if

$$N = \sum_{i' \in [n]} p_{i'} \cdot \sum_{i' \in [n]} q_{i'}$$
$$\wedge \quad 0 \le p_{i'} < M \quad \wedge \quad 0 \le q_{i'} < M \qquad \text{for all } i' \in [n]$$

where the sums and product are taken over the integers.[b]

- 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 the parties halt successfully and output $(\texttt{biprime}, \mathsf{sid}, N)$ to the environment; otherwise, they 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 and output $(\texttt{non-biprime}, \mathsf{sid})$ to the environment. 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.

[b] Note that computations over the (unbounded) integers are technically inexpressible as circuits, as in practice this predicate must be expressed. Thus, in practice, the predicate must instead use a modulus sufficiently large to make overflow impossible, given the input-size constraints.

4.4 Security Sketches

We now informally argue that π_{RSAGen} realizes $\mathcal{F}_{\mathsf{RSAGen}}$ in the semi-honest and malicious models.

Theorem 4.5. *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 Sketch. In lieu of arguing for the correctness of our protocol, we refer the reader to the explanation in Section 4.1, and focus here on the strategy of a simulator $\mathcal{S}$ against a semi-honest adversary $\mathcal{A}$ who corrupts the parties indexed by $\mathbf{P}^*$. $\mathcal{S}$ forwards all messages between $\mathcal{A}$ and the environment faithfully.

In Step 1 of π_{RSAGen}, for each $j \in [2, \ell]$, $\mathcal{S}$ receives the `sample` instruction with modulus $\mathbf{m}_j$ on behalf of $\mathcal{F}_{\mathsf{AugMul}}$ from all parties indexed by $\mathbf{P}^*$. For each j it then samples $(\mathbf{p}_{i,j}, \mathbf{q}_{i,j}, \mathbf{N}_{i,j}) \leftarrow \mathbb{Z}^3_{\mathbf{m}_j}$ uniformly for $i \in \mathbf{P}^*$, and returns each triple to the appropriate party.

Step 2 involves no interaction on the part of the parties, but it is at this point that $\mathcal{S}$ computes p_i and q_i for $i \in \mathbf{P}^*$, in the same way that the parties themselves do. Note that since $\mathbf{p}_{*,1}$ and $\mathbf{q}_{*,1}$ are deterministically chosen, they are known to $\mathcal{S}$. The simulator then sends these shares to $\mathcal{F}_{\mathsf{RSAGen}}$ via the functionality's `adv-input` interface, and receives in return either a biprime N, or two factors p and q such that $N \coloneqq p \cdot q$ is not a biprime. Regardless, it instructs $\mathcal{F}_{\mathsf{RSAGen}}$ to `proceed`.

In Step 3 of π_{RSAGen}, $\mathcal{S}$ receives two `input` instructions from each corrupted party for each $j \in [\ell+1, \ell']$ on behalf of $\mathcal{F}_{\mathsf{AugMul}}$, and confirms receipt as $\mathcal{F}_{\mathsf{AugMul}}$ would. Subsequently, for each $j \in [\ell+1, \ell']$, the corrupt parties all send a `multiply` instruction, and then $\mathcal{S}$ samples $\mathbf{N}_{i,j} \leftarrow \mathbb{Z}_{\mathbf{m}_j}$ for $i \in [n]$ subject to

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

and returns each share to the matching corrupt party.

In Step 4 of π_{RSAGen}, for every $j \in [\ell']$, every corrupt party $\mathcal{P}_{i'}$ for $i' \in \mathbf{P}^*$, and every honest party $\mathcal{P}_i$ for $i \in [n] \setminus \mathbf{P}^*$, $\mathcal{S}$ sends $\mathbf{N}_{i,j}$ to $\mathcal{P}_{i'}$ on behalf of $\mathcal{P}_i$, and receives $\mathbf{N}_{i',j}$ (which it already knows) in reply.

To simulate the final steps of π_{RSAGen}, $\mathcal{S}$ tries to divide N by all primes smaller than B. If it succeeds, then the protocol is complete. Otherwise, it receives `check-biprimality` from all of the corrupt parties on behalf of $\mathcal{F}_{\mathsf{Biprime}}$, and replies with `biprime` or `not-biprime` as appropriate. It can be verified by inspection that the view of the environment is identically distributed in the ideal-world experiment containing $\mathcal{S}$ and honest parties that interact with $\mathcal{F}_{\mathsf{RSAGen}}$, and the real-world experiment containing $\mathcal{A}$ and parties running π_{RSAGen}. □

Theorem 4.6. *If factoring biprimes sampled by* BFGM *is hard, then* π_{RSAGen} *UC-realizes* $\mathcal{F}_{\mathsf{RSAGen}}$ *in the* $(\mathcal{F}_{\mathsf{AugMul}}, \mathcal{F}_{\mathsf{Biprime}})$*-hybrid model against a static, malicious PPT 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 its inputs to $\mathcal{F}_{\mathsf{AugMul}}$ or $\mathcal{F}_{\mathsf{Biprime}}$, then the simulator can follow the same strategy as in the semi-honest case. At any point if the adversary deviates from the protocol, the simulator requests $\mathcal{F}_{\mathsf{RSAGen}}$ to reveal all honest parties' shares, and thereafter the simulator uses them by effectively running the code of the 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 the honest parties must only detect cheating when N is truly a biprime and the adversary has sabotaged a successful candidate; if N is not a biprime and would have been rejected anyway, then cheat-detection is unimportant. We analyze all possible cases where the adversary deviates from the protocol below. Let N be defined as the value implied by parties' sampled shares in Step 1 of π_{RSAGen}.

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 then the incorrect reconstruction will be caught by the explicit secure predicate check during the consistency-check phase. If N' is a non-biprime then the 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 events will occur:

- $\mathcal{F}_{\mathsf{Biprime}}$ rejects this candidate. In this case, all parties reveal their shares of p and q to one another (with guaranteed correctness via $\mathcal{F}_{\mathsf{AugMul}}$) and locally test their primality. This will reveal 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). Because N only has two factors, there is exactly one pair of inputs that the adversary can supply to $\mathcal{F}_{\mathsf{Biprime}}$ to induce this scenario, apart from the pair specified by the protocol. Finding this alternative pair of satisfying inputs implies factoring N. We are careful to rely on the hardness of factoring only in this case, where by premise N 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 then the incorrect reconstruction will be caught during the consistency-check phase, just as when N is a biprime. If N' is a non-biprime then 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', with the implication that some party has cheated.

Thus the adversary is always caught when trying to sabotage a true biprime, and it can never sneak a non-biprime past the consistency check. Because the real-world protocol always aborts in the case of cheating, it is indistinguishable from the simulation described above, assuming that factoring is hard. □

5 Distributed Biprimality Testing

In this section, we present protocols realizing $\mathcal{F}_{\mathsf{Biprime}}$. In Section 5.1, we discuss the semi-honest setting, and in Section 5.2, the malicious setting.

5.1 The Semi-Honest Setting

In the semi-honest setting, $\mathcal{F}_{\mathsf{Biprime}}$ can be realized by the biprimality-testing protocol of Boneh and Franklin [BF01]. Loosely speaking, the Boneh-Franklin protocol is a variant of the Miller-Rabin test: for a randomly chosen $\gamma \in \mathbb{Z}_N^*$ with Jacobi symbol 1, it checks whether $\gamma^{(N-p-q+1)/4} \equiv \pm 1 \pmod{N}$ (recall that $\varphi(N) = N - p - q + 1$). A biprime will always pass this test, but non-biprimes may yield a false positive with probability 1/2. The test is repeated s times (either sequentially or concurrently) in order to bound the probability of proceeding with a false positive to 2^{-s} (where s is a statistical parameter).

The above test filters out all non-biprimes *except* those with factors of the form $p = a_1^{b_1}$ and $q = a_2^{b_2}$, with $q \equiv 1 \pmod{a_1^{b_1-1}}$. This final class of non-biprimes is filtered by securely sampling $r \leftarrow \mathbb{Z}_N$, computing $z := r \cdot (p+q-1)$, and then testing whether $\gcd(z, N) = 1$[7]. Boneh and Franklin suggest that the secure sampling of r and the computation of z can be done via generic MPC; we provide a functionality $\mathcal{F}_{\mathsf{ComCompute}}$ (see Appendix A.2) that is adequate for the task, but note that there are more efficient methods. Regardless, we note that this final test does induce some false negatives (modeled by BFGM), and refer

[7] This is accomplished by testing $\gcd(z \pmod{N}, N) = 1$, which is equivalent as any factor of z and N also divides $z \pmod{N}$.

the reader to Boneh and Franklin [BF01, Section 4.1] for a more comprehensive discussion. The following lemma follows immediately from their work.

Lemma 5.1. *The biprimality-testing protocol described by Boneh and Franklin [BF01] UC-realizes $\mathcal{F}_{\mathsf{Biprime}}$ with statistical security 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

Unlike a semi-honest adversary, we permit a malicious adversary to force a true biprime to fail our biprimality test, and detect such behavior using independent mechanisms in the π_{RSAGen} protocol. However, we must ensure that a non-biprime can never pass the test with more than negligible probability. To achieve this, we use a derivative of the biprimality-testing protocol of Frederiksen et al. [FLOP18]; relative to their protocol, ours is simpler, and we prove that it UC-realizes $\mathcal{F}_{\mathsf{Biprime}}$.

The protocol essentially comprises a randomized version of the semi-honest Boneh-Franklin test described previously, followed by a Schnorr-like protocol to verify that the test was performed correctly. The soundness error of the underlying biprimality test is compounded by the Schnorr-like protocol's soundness error to yield a combined error of $3/4$; this necessitates an increase in the number of iterations by a factor of $\log_{4/3}(2) < 2.5$. While this is sufficient to ensure the test itself is carried out honestly, it does not ensure the correct inputs are used. Consequently, generic MPC is used to verify the relationship between the messages involved in the Schnorr-like protocol and the true candidate given by N and shares of its factors. As a side effect, this generic computation samples $r \leftarrow \mathbb{Z}_N$ and outputs $z = r \cdot (p+q-1) \bmod N$ so that the GCD test can afterward be run locally by each party.

Our protocol makes use of a number of subfunctionalities, all of which are standard and described in Appendix A.2. Namely, we use a coin-tossing functionality $\mathcal{F}_{\mathsf{CT}}$ to uniformly sample an element from some set, the one-to-many commitment functionality $\mathcal{F}_{\mathsf{Com}}$, the generic MPC functionality over committed inputs $\mathcal{F}_{\mathsf{ComCompute}}$, and the integer-sharing-of-zero functionality $\mathcal{F}_{\mathsf{Zero}}$. In addition, the protocol uses the algorithm $\mathsf{VerifyBiprime}$ (Algorithm 5.3).

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

This protocol is parametrized by the number of parties n and an integer M. In addition, there is a statistical parameter s. The parties have access to the $\mathcal{F}_{\mathsf{CT}}$, $\mathcal{F}_{\mathsf{Com}}$, $\mathcal{F}_{\mathsf{ComCompute}}$, and $\mathcal{F}_{\mathsf{Zero}}$ functionalities.

Input Commitment:

1. Upon receiving input $(\texttt{check-biprimality}, \mathsf{sid}, N, p_i, q_i)$ from the environment, each party $\mathcal{P}_i$ for $i \in [n]$ samples $\boldsymbol{\tau}_{i,j} \leftarrow \mathbb{Z}_{M\cdot 2^{s+1}}$ for $j \in [2.5s]$ and commits to these values, along with its shares of p and q, by sending $(\texttt{commit}, \mathsf{GenSID}(\mathsf{sid}, i), (p_i, q_i, \boldsymbol{\tau}_{i,*}))$ to $\mathcal{F}_{\mathsf{ComCompute}}(n)$.

Boneh-Franklin Test:

2. Each party $\mathcal{P}_i$ for $i \in [n]$ sends $(\texttt{sample}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{Zero}}(n, 2^{2\kappa+s})$ and receives $(\texttt{zero-share}, \mathsf{sid}, r_i)$ in response.

3. For $j \in [2.5s]$, the parties invoke $\mathcal{F}_{\mathsf{CT}}(n, \mathbb{J}_N)$, where $\mathbb{J}_N$ is the subdomain of $\mathbb{Z}_N^*$ that contains only values with Jacobi symbol 1. The parties define vector $\boldsymbol{\gamma}$ that contains the $2.5s$ sampled values.

4. For every $j \in [2.5s]$, party $\mathcal{P}_1$ computes[a]

$$\boldsymbol{\chi}_{1,j} := \boldsymbol{\gamma}_j^{r_1-(p_1+q_1-6)/4} \mod N$$

and every other party $\mathcal{P}_i$ for $i \in [2, n]$ computes

$$\boldsymbol{\chi}_{i,j} := \boldsymbol{\gamma}_j^{r_i-(p_i+q_i)/4} \mod N$$

5. Every $\mathcal{P}_i$ for $i \in [n]$ sends $(\texttt{commit}, \mathsf{GenSID}(\mathsf{sid}, i), \boldsymbol{\chi}_{i,*}, [n])$ to $\mathcal{F}_{\mathsf{Com}}(n)$.

6. After being notified that all other parties are committed, each party $\mathcal{P}_i$ for $i \in [n]$ sends $(\texttt{decommit}, \mathsf{GenSID}(\mathsf{sid}, i))$ to $\mathcal{F}_{\mathsf{Com}}(n)$, and in response receives $\boldsymbol{\chi}_{i',*}$ from $\mathcal{F}_{\mathsf{Com}}(n)$ for $i' \in [n] \setminus \{i\}$.

7. The parties output $(\texttt{not-biprime}, \mathsf{sid})$ to the environment and halt if there exists $j \in [2.5s]$ such that

$$\boldsymbol{\gamma}_j^{(N-5)/4} \cdot \prod_{i \in [n]} \boldsymbol{\chi}_{i,j} \not\equiv \pm 1 \pmod N$$

Consistency Check and GCD Test:

8. For $j \in [2.5s]$, each party $\mathcal{P}_i$ for $i \in [n]$ computes $\boldsymbol{\alpha}_{i,j} := \boldsymbol{\gamma}_j^{\boldsymbol{\tau}_{i,j}} \mod N$. The parties all broadcast the values they have computed to one another.

9. The parties all send $(\texttt{flip}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{CT}}(n, \{0,1\}^{2.5s})$ to obtain an agreed-upon random bit vector $\mathbf{c}$ of length $2.5s$.

10. For $j \in [2.5s]$, party $\mathcal{P}_1$ computes $\boldsymbol{\zeta}_{1,j} := \boldsymbol{\tau}_{1,j} - \mathbf{c}_j \cdot (p_1+q_1)/4$, and every other party $\mathcal{P}_i$ for $i \in [2, n]$ computes $\boldsymbol{\zeta}_{i,j} := \boldsymbol{\tau}_{i,j} - \mathbf{c}_j \cdot (p_i + q_i - 6)/4$. They all broadcast the values they have computed to one another.

11. The parties abort if there exists any $j \in [2.5s]$ such that

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

12. Let C be a circuit computing $\mathsf{VerifyBiprime}(N, M, \mathbf{c}, \{\cdot, \cdot, \cdot, \boldsymbol{\zeta}_{i,*}\}_{i\in[n]})$; that is, let it be a circuit representation of Algorithm 5.3 with the public values N, M, $\mathbf{c}$, and $\boldsymbol{\zeta}$ hardcoded. The parties send $(\texttt{compute}, \mathsf{sid}, \{\mathsf{GenSID}(\mathsf{sid}, i)\}_{i\in[n]}, C)$ to $\mathcal{F}_{\mathsf{ComCompute}}(n)$, and in response they all receive $(\texttt{result}, \mathsf{sid}, z)$. If $z = \perp$ then the parties abort.

13. The parties terminate and output $(\texttt{biprime}, \mathsf{sid})$ to the environment if $\gcd(z, N) = 1$, or terminate and output $(\texttt{not-biprime}, \mathsf{sid})$ otherwise.

[a] Recall that $p_1 \equiv q_1 \equiv 3 \pmod 4$, and so subtracting 6 from their sum ensures that division by 4 can be performed without computing a modular multiplicative inverse in $\mathbb{Z}_N^*$. We compensate for this offset using another offset in Step 7.

Below we present the algorithm VerifyBiprime that is used for the GCD test. The inputs are the candidate biprime N, an integer M (the bound on the shares' size), a bit-vector $\mathbf{c}$ of length $2.5s$, and for each $i \in [n]$ a tuple consisting of the shares p_i and q_i with the Schnorr-like messages $\boldsymbol{\tau}_{i,*}$ and $\boldsymbol{\zeta}_{i,*}$ generated by $\mathcal{P}_i$. The algorithm verifies that all input values are compatible, and returns $z = r \cdot (p + q - 1) \bmod N$ for a random r.

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

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

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

2. Return z if and only if it holds that

$$\begin{aligned} N = \sum_{i\in[n]} p_i \cdot \sum_{i\in[n]} q_i & \\ \wedge\quad 0 \le p_i < M \quad \wedge \quad 0 \le q_i < M & \qquad \text{for all } i \in [n] \\ \wedge\quad \boldsymbol{\tau}_{1,j} = \boldsymbol{\zeta}_{1,j} + \mathbf{c}_j \cdot (p_1 + q_1 - 6)/4 & \qquad \text{for all } j \in [2.5s] \\ \wedge\quad \boldsymbol{\tau}_{i,j} = \boldsymbol{\zeta}_{i,j} + \mathbf{c}_j \cdot (p_i + q_i)/4 & \qquad \text{for all } i \in [2, n] \text{ and } j \in [2.5s] \end{aligned}$$

If any part of the above predicate does not hold, output $\perp$.

Theorem 5.4. *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 with statistical security against a static, malicious adversary that corrupts up to $n - 1$ parties.*

Proof Sketch. Our simulator $\mathcal{S}$ for $\mathcal{F}_{\mathsf{Biprime}}$ receives N as common input. Let $\mathbf{P}^*$ and $\overline{\mathbf{P}^*}$ be vectors indexing the corrupt and honest parties, respectively. To simulate Steps 1 through 3 of π_{Biprime}, $\mathcal{S}$ simply behaves as $\mathcal{F}_{\mathsf{CT}}$, $\mathcal{F}_{\mathsf{Zero}}$, and $\mathcal{F}_{\mathsf{ComCompute}}$ would in its interactions with the corrupt parties on their behalf, remembering the values received and transmitted. Before continuing, $\mathcal{S}$ submits the corrupted parties' shares of p and q to $\mathcal{F}_{\mathsf{Biprime}}$ on their behalf. In response,

$\mathcal{F}_{\mathsf{Biprime}}$ either informs $\mathcal{S}$ that N is a biprime, or leaks the honest parties' shares. In Step 4, $\mathcal{S}$ again behaves exactly as $\mathcal{F}_{\mathsf{Com}}$ would. During the remainder of the protocol, the simulator must follow one of two different strategies, conditioned on whether or not N is a biprime. We will show that both strategies lead to a simulation that is statistically indistinguishable from the real-world experiment.

- If $\mathcal{F}_{\mathsf{Biprime}}$ reported that N is a biprime, then we know by the specification of $\mathcal{F}_{\mathsf{Biprime}}$ that the corrupt parties committed to correct shares of p and q in Step 1 of π_{Biprime}. Boneh and Franklin [BF01] showed that the value (i.e., sign) of the right-hand side of the equality in Step 7 is predictable and related to the value of $\boldsymbol{\gamma}_j$. We refer to them for a precise description and proof. If without loss of generality we take that value to be 1, then $\mathcal{S}$ can simulate iteration j of Steps 6 and 7 as follows. First, $\mathcal{S}$ computes $\hat{\boldsymbol{\chi}}_{i,j}$ for $i \in \mathbf{P}^*$ to be the corrupt parties' ideal values of $\boldsymbol{\chi}_{i,j}$ as defined in Step 4 of π_{Biprime}. Then, $\mathcal{S}$ samples $\boldsymbol{\chi}_{i,j} \leftarrow \mathbb{Z}_N^*$ uniformly for $i \in \overline{\mathbf{P}^*}$ subject to

  $$\prod_{i \in \overline{\mathbf{P}^*}} \boldsymbol{\chi}_{i,j} \equiv \frac{\boldsymbol{\gamma}_j^{(5-N)/4}}{\prod\limits_{i \in \mathbf{P}^*} \hat{\boldsymbol{\chi}}_{i,j}} \pmod{N}$$

  and simulates Step 6 by releasing $\boldsymbol{\chi}_{i,j}$ for $i \in \overline{\mathbf{P}^*}$ to the corrupt parties on behalf of $\mathcal{F}_{\mathsf{Com}}$. These values are statistically close to their counterparts in the real protocol. Finally, $\mathcal{S}$ simulates Step 7 by running the test for itself and sending the `cheat` command to $\mathcal{F}_{\mathsf{Biprime}}$ on failure.

  Given the information now known to $\mathcal{S}$, Steps 8 through 11 of π_{Biprime} can be simulated in a manner similar to the simulation of a common Schnorr protocol: $\mathcal{S}$ simply chooses $\boldsymbol{\zeta}_{i,*} \leftarrow \mathbb{Z}_{M \cdot 2^{s+1}}^{2.5s}$ uniformly for $i \in \overline{\mathbf{P}^*}$, fixes $\mathbf{c} \leftarrow \{0,1\}^{2.5s}$ ahead of time, and then works backwards via the equation in Step 11 to compute the values of $\boldsymbol{\alpha}_{i,*}$ for $i \in \overline{\mathbf{P}^*}$ that it must send on behalf of the honest parties in Step 8. These values are statistically close to their counterparts in the real protocol.

  $\mathcal{S}$ finally simulates the remaining steps of π_{Biprime} by checking the VerifyBiprime predicate itself (since the final GCD test is purely local, no action need be taken by $\mathcal{S}$). If at any point after Step 4 the corrupt parties have cheated (i.e., sent an unexpected value or violated the VerifyBiprime predicate), then $\mathcal{S}$ instructs $\mathcal{F}_{\mathsf{Biprime}}$ to abort. Otherwise, it sends the `proceed` command to $\mathcal{F}_{\mathsf{Biprime}}$, completing the simulation.

- If $\mathcal{F}_{\mathsf{Biprime}}$ reported that N is *not* a biprime (which may indicate that the corrupt parties supplied incorrect shares of p or q), then it also leaked the honest parties' shares of p and q to $\mathcal{S}$. Thus, $\mathcal{S}$ can simulate Steps 4 through 13 of π_{Biprime} by running the honest parties' code on their behalf, and instructing $\mathcal{F}_{\mathsf{Biprime}}$ to abort if the honest parties' code indicates that they should do so. In the ideal-world experiment defined by this simulation strategy, the distribution of aborts is exactly the same as the distribution of aborts in the real-world experiment. In all *non-aborting* instances of the ideal-world experiment, the honest parties report to the environment that N is a non-biprime. Thus, we need only prove that there is no strategy by which the corrupt parties can successfully convince the honest parties that N is a biprime in the real world.

In order to get away with such a real-world cheat, the adversary must cheat in every iteration j of Steps 4 through 6 for which

$$\boldsymbol{\gamma}_j^{(N-p-q)/4} \not\equiv \pm 1 \pmod{N}$$

Specifically, in every such iteration j, the corrupt parties must contrive to send values $\boldsymbol{\chi}_{i,j}$ for $i \in \mathbf{P}^*$ such that

$$\boldsymbol{\gamma}_j^{(N-5)/4} \cdot \prod_{i\in[n]} \boldsymbol{\chi}_{i,j} \equiv \boldsymbol{\gamma}_j^{(N-p-q)/4+\boldsymbol{\Delta}_{1,j}} \equiv \pm 1 \pmod{N}$$

for some nonzero offset value $\boldsymbol{\Delta}_{1,j}$. We can define a similar offset $\boldsymbol{\Delta}_{2,j}$ for the corrupt parties' transmitted values of $\boldsymbol{\alpha}_{i,j}$, relative to the values of $\boldsymbol{\tau}_{i,j}$ committed in Step 1:

$$\boldsymbol{\gamma}_j^{\boldsymbol{\Delta}_{2,j}} \cdot \prod_{i\in[n]} \boldsymbol{\alpha}_{i,j} \equiv \prod_{i\in[n]} \boldsymbol{\gamma}_j^{\boldsymbol{\tau}_{i,j}} \pmod{N}$$

Since we have presupposed that no abort is triggered, we know that the corrupt parties *must* transmit correctly calculated values of $\boldsymbol{\zeta}_{i,*}$ in Step 10 of π_{Biprime}, or else Step 12 would abort when these values are checked by the $\mathsf{VerifyBiprime}$ predicate. It follows from this fact and from the equation in Step 11 that $\boldsymbol{\Delta}_{2,j} \equiv \mathbf{c}_j \cdot \boldsymbol{\Delta}_{1,j} \pmod{\varphi(N)}$, where $\varphi(\cdot)$ is Euler's totient function. However, both $\boldsymbol{\Delta}_{1,*}$ and $\boldsymbol{\Delta}_{2,*}$ are fixed before $\mathbf{c}$ is revealed to the corrupt parties, and so the adversary can succeed in this cheat with probability at most $1/2$ for any individual iteration j.

Per Boneh and Franklin [BF01, Lemma 4.1], a particular iteration j of Steps 4 through 6 of π_{Biprime} produces a false positive result with probability at most $1/2$ if the adversary behaves honestly. If we assume that the adversary cheats always and only when a false positive would not have been produced by honest behavior, then the total probability of an adversary producing a positive outcome in the j^{th} iteration of Steps 4 through 6 is upper-bounded by $3/4$. The probability that an adversary succeeds over all $2.5s$ iterations is therefore at most $(3/4)^{2.5s} < 2^{-s}$. Thus, the adversary has a negligible chance to force the acceptance of a non-biprime in the real world, and the distribution of outcomes produced by $\mathcal{S}$ is statistically indistinguishable from to the real-world distribution. □

6 Efficiency Analysis

In this section we give both an asymptotic and a closed-form concrete cost analysis of our protocol, with both semi-honest and malicious security. We begin by addressing the success probability of our ideal functionality $\mathcal{F}_{\mathsf{RSAGen}}$, as determined by analysis of the sampling function $\mathsf{CRTSample}$ that it uses. We also discuss various strategies for composing invocations of $\mathcal{F}_{\mathsf{RSAGen}}$ to amplify the probability that a biprime is successfully produced. After this, we analyze the costs of the protocols realizing $\mathcal{F}_{\mathsf{AugMul}}$ and $\mathcal{F}_{\mathsf{Biprime}}$ in Section 6.2, and in Section 6.3, we compose those costs to give an overall cost for each invocation of π_{RSAGen}. In Section 6.4, we discuss a compositional strategy that leads to constant or expected-constant rounds overall, and calculate the precise number of rounds required. Finally, in Section 6.5, we provide a performance comparison to the protocol of Frederiksen et al. [FLOP18].

6.1 Per-Instance Success Probability

By construction, the success probability of $\mathcal{F}_{\mathsf{RSAGen}}$ is identical to that of $\mathsf{CRTSample}$. To bound the success probability of $\mathsf{CRTSample}$, it suffices to determine the probability that a randomly chosen value is prime, conditioned on that value having $\mathbf{m}$-coprimality (per Definition 3.5) relative to the (κ, n)-near-primorial vector $\mathbf{m}$ of length ℓ. We begin by bounding the probability of finding a prime from below, relative to $\max(\mathbf{m})$, the largest value in $\mathbf{m}$.

Lemma 6.1 ([BF01])**.** *Given a (κ, n)-near-primorial vector $\mathbf{m}$,*

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

Observe that this is a *concrete* lower bound. Next, in the interest of asymptotic analysis, we bound the value of the maximum element in $\mathbf{m}$. We first adapt a lemma from Rosser and Schoenfeld [RS62] to bound $\max(\mathbf{m})$ with respect to the product of the elements of $\mathbf{m}$, and we then use this to construct a bound with respect to κ.

Lemma 6.2 ([RS62], Theorem 4)**.** *Given a (κ, n)-near-primorial vector $\mathbf{m}$ and its product M, it holds that* $\max(\mathbf{m}) \in \Theta(\log M)$*.*

Lemma 6.3. *Given a (κ, n)-near-primorial vector $\mathbf{m}$, it holds that* $\max(\mathbf{m}) \in \Omega(\kappa)$*.*

Proof. Let M be the product of $\mathbf{m}$. By Definition 3.4, 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 $M \in \Omega(2^{\kappa - \log n - \log \kappa})$. Combining this with Lemma 6.2 yields $\max(\mathbf{m}) \in \Omega(\kappa - \log n - \log \kappa)$ and finally, Lemma 3.7 constrains $2 \leq n < \kappa$, which yields $\max(\mathbf{m}) \in \Omega(\kappa)$. □

Combining Lemmas 6.1 and 6.3 and considering that $\mathsf{CRTSample}$ must sample two primes simultaneously (but independently) yields an asymptotic figure for the success probability of $\mathsf{CRTSample}$. Since this governs the number of times $\mathcal{F}_{\mathsf{RSAGen}}$ must be invoked in order to generate a biprime, we refer to it as our sampling-efficiency theorem.

κ	ℓ	ℓ'	$\max(\mathbf{m})$	$\Pr[\text{Success}]$
Asymptotic	$O(\kappa/\log\kappa)$	$O(\kappa/\log\kappa)$	$\Omega(\kappa)$	$\Omega(\log^2\kappa/\kappa^2)$
1024	130	233	739	$\geq 1/3607$
1536	182	327	1093	$\geq 1/7250$
2048	231	418	1459	$\geq 1/11832$

Table 6.6: CRT-form Sampling Parameters, along with per-iteration success probabilities. Recall that biprimes produced are of size 2κ. $\mathbf{m}$ is the (κ, n)-near-primorial vector, with $\ell = |\mathbf{m}|$. ℓ' is the length of the extended vector required by our protocol, as described in Definition 4.3. Note that the number of parties n has no concrete effect on these values.

Theorem 6.4 (Sampling-Efficiency Theorem)**.** *For any strict subset $\mathbf{P}^* \subset [n]$ and any $(p_i, q_i) \in \mathbb{Z}^2_{2^\kappa}$ for $i \in \mathbf{P}^*$, $\mathsf{CRTSample}(n, \kappa, \{(p_i, q_i)\}_{i\in\mathbf{P}^*})$ samples a biprime with probability $\Omega(\log^2\kappa/\kappa^2)$.*

In order to understand the concrete efficiency of our approach, we determined the unique (κ, n)-near-primorial vectors corresponding to several common RSA security parameter values, and then used Lemma 6.1 to derive concrete success probabilities. These are reported in Table 6.6. For completeness, we also report the asymptotic size of the (κ, n)-near-primorial vector; this can be derived by combining Lemma 6.2 with the following lemma:

Lemma 6.5 ([RS62], Theorem 2)**.** *Given a (κ, n)-near-primorial vector $\mathbf{m}$ of length ℓ, it holds that $\ell \in \Theta\left(\max(\mathbf{m})/\log\max(\mathbf{m})\right)$.*

Compositional Strategies. As an immediate corollary to Theorem 6.4, the expected number of invocations of $\mathcal{F}_{\mathsf{RSAGen}}$ required to produce a biprime is in $O(\kappa^2/\log^2\kappa)$. Concretely, this corresponds to 3607 invocations when $\kappa = 1024$, or 11832 invocations when $\kappa = 2048$. As an alternative to sequential invocation, $\mathcal{F}_{\mathsf{RSAGen}}$ can be invoked concurrently in batches tuned for a desired probability of success. $O(\rho/(\log\kappa - \log(\kappa^2 - \log^2\kappa)))$ concurrent invocations are required to sample a biprime with probability at least $1-2^{-\rho}$. Concretely, with $\rho = 40$, this corresponds to roughly 100000 invocations for $\kappa = 1024$, or 330000 invocations for $\kappa = 2048$. Two sequential batches of concurrent invocations are required in expectation if $\rho = 1$, with roughly 2500 invocations of $\mathcal{F}_{\mathsf{RSAGen}}$ per batch for $\kappa = 1024$, or 8200 invocations per batch for $\kappa = 2048$.

6.2 The Cost of Instantiating $\mathcal{F}_{\mathsf{Biprime}}$ and $\mathcal{F}_{\mathsf{AugMul}}$

Before we can discuss the concrete costs of our main sampling protocol, we must determine the costs of instantiating the functionalities it uses. Since $\mathcal{F}_{\mathsf{Biprime}}$ was specifically formulated to model the biprimality-testing protocol of Boneh and

Franklin [BF01], it is natural to assume we use that protocol (or our malicious-secure extension) to instantiate it. On the other hand, in both the semi-honest and malicious settings, many sensible approaches exist for instantiating $\mathcal{F}_{\mathsf{AugMul}}$. We choose to base our instantiation on oblivious transfer in both settings. Although OT-based multiplication is not the most bandwidth-efficient option, it compares favorably to alternatives in the real world [DKLS18, DKLS19], and OT can be built assuming only the hardness of factoring, if desired [EGL85]. We also assume certain reasonable practical concessions are made. For example, we assume that $\mathcal{F}_{\mathsf{Com}}$ and $\mathcal{F}_{\mathsf{CT}}$ are both implemented via a Random Oracle. We note that under these instantiations, committing a value requires 2λ bits to be broadcast,[8] and $\mathcal{F}_{\mathsf{CT}}$ requires no communication at all. We also assume $\mathcal{F}_{\mathsf{Zero}}$ requires no communication: a trivial modification of the Pseudorandom Zero-Sharing protocol of Cramer et al. [CDI05] yields a protocol that realizes $\mathcal{F}_{\mathsf{Zero}}$ against active adversaries with no interaction in the $\mathcal{F}_{\mathsf{CT}}$-hybrid model.

Broadcast. The standard practice of practical implementations of MPC protocols (e.g., [HSS17, WRK17]) uses the simple echo broadcast of Goldwasser and Lindell [GL05] for instantiating the broadcast channel. The resulting protocol only guarantees *non-unanimous* abort (i.e., each honest party either obtains the correct output or locally aborts, but there is no global agreement on abort), but preserves all correctness and privacy guarantees. This approach doubles the round count and adds $(n-1)\cdot\lambda$ bits per party per original round to the communication cost, because each party is required to send a hash of the broadcast messages received in one round to all other parties before the next round.

In our setting, we can take an even simpler approach and run the protocol optimistically over point-to-point channels. At the end, every party hashes the entire transcript of broadcast-messages and sends the digest to all other parties. If the digests do not agree, the party aborts; otherwise, the party terminates with the output value of the protocol. Therefore, for the sake of calculating concrete costs, we consider the cost of broadcasting a bit to be the same as the cost of transmitting it to all parties (apart from the sender) individually.

Oblivious Transfer. We give concrete efficiency figures assuming that Correlated OT (i.e., OT in which the sender chooses a correlation between two messages, instead of two independent messages) is used in place of standard OT, and assuming that it is instantiated via one of two specific OT-extension protocols. The more efficient of the two is the recently introduced Silent OT-extension protocol of Boyle et al. [BCG$^+$19]. When used as a Correlated OT-extension on a correlation from the field $\mathbb{Z}_m$, it incurs a cost of $(|m|+1)/2$ bits transmitted per party (averaged over the sender and receiver), over two rounds. However, it requires a variant of the Learning Parity with Noise (LPN) assumption, and it has a large one-time setup cost.

As an alternative, the OT-extension protocol of Keller et al. [KOS15] (hereafter referred to as "KOS OT") assumes only a base OT functionality, and has

[8] Where λ is a computational security parameter as described in Section 2.

a much cheaper one-time setup. When used as a Correlated OT-extension on a correlation from the field $\mathbb{Z}_m$, this protocol incurs a cost of $(|m| + \lambda)/2$ bits transmitted per party (averaged over the sender and receiver) over two rounds. In addition to this cost the sender must pay an overhead cost for each *batch* of OTs performed, where a batch may comprise polynomially many individual OTs with an arbitrary mixture of correlation lengths. Since this overhead does not depend on the total number of OTs or the correlation lengths, it can be amortized into irrelevance if the OTs in our protocol can be arranged into a small number of batches. Since this is indeed the case, we ignore the overhead cost for the remainder of our cost analysis. For the sake of analyzing concrete efficiency, we also ignore the one-time setup costs of both OT-extension protocols.

Both OT-extension protocols attain malicious security, but KOS OT has little overhead relative to the best semi-honest protocols, and Silent OT is the most efficient currently known OT protocol of any kind. Consequently, we use them in the semi-honest setting as well.

$\mathcal{F}_{\mathsf{AugMul}}$ in the Semi-Honest Setting. In the semi-honest setting, parties can be trusted to reuse inputs when instructed to do so, and the predicate check and input-revelation interfaces of $\mathcal{F}_{\mathsf{AugMul}}$ need not be implemented, since π_{RSAGen} will never call them in the semi-honest setting. Consequently, an extremely simple protocol suffices to realize $\mathcal{F}_{\mathsf{AugMul}}$ against semi-honest adversaries. When the parties wish to use the `multiply` interface with a modulus m, they simply engage in instances of Gilboa's classic OT-multiplication protocol [Gil99], arranged as per the GMW multiplication paradigm [GMW87] to form a multiparty multiplier. This implies $2|m|$ oblivious transfers per pair of parties, all with $|m|$-bit messages. If Silent OT is used, the total cost is $(n-1)\cdot|m|\cdot(|m|+1)$ bits transmitted per party (averaged over all parties), over two rounds. If KOS OT is used, the total cost is instead $(n-1)\cdot|m|\cdot(|m|+\lambda)$ bits. Note that this is substantially worse if $|m|$ is small.

The `sample` interface of $\mathcal{F}_{\mathsf{AugMul}}$ can be realized using the same multiplication protocol in a natural way: the parties simply choose random shares of two values and multiply those values together. They multiply their output shares together with another randomly sampled set of shares, and then broadcast their shares of this second product. They repeat the whole process if the second product is found to be zero, or otherwise take their shares of the first product as their outputs. Each iteration of this sampling procedure succeeds with probability $(m-1)^3/m^3$. If iterations are run concurrently in batches of size c, then the expected number of batches is $b = m^3/(m^3 - 3c\cdot m^2 + 3c\cdot m - c)$. Given this value b, the expected total cost is $(n-1)\cdot b\cdot c\cdot|m|\cdot(2|m|+3)$ bits transmitted per party (on average) if Silent OT is used, or $(n-1)\cdot b\cdot c\cdot|m|\cdot(2|m|+2\lambda+1)$ bits if KOS OT is used. The number of rounds is $5b$ in expectation.

$\mathcal{F}_{\mathsf{Biprime}}$ in the Semi-Honest Setting. To realize $\mathcal{F}_{\mathsf{Biprime}}$ in the semi-honest setting, we use the Boneh-Franklin biprimality test [BF01]. In this protocol, the parties begin by participating in up to s iterations of a test in which they each broadcast a value of size 2κ to all other parties. In the bandwidth-optimal protocol variant, these iterations are sequential. Each has a soundness error of $1/2$, and so the expected number of iterations is two if N is not a biprime, yielding a concrete bandwidth cost for this first step of $2(n-1) \cdot s \cdot \kappa$ bits per party if N is a biprime, or $4(n-1) \cdot \kappa$ bits per party in expectation otherwise. In the round-optimal protocol variant, these iterations are concurrent, and the cost is always $2(n-1) \cdot s \cdot \kappa$ bits per party, over one round.

If all iterations of the previous test succeed, then the parties perform the GCD Test: they securely multiply two shared values modulo N, and reveal the output. Whereas Boneh and Franklin recommend the BGW generic MPC protocol for this task, we will instead assume that Gilboa's OT-multiplication protocol is used, in a GMW-like arrangement as previously described, followed by an exchanging of output shares. All shares in one of the two input sets are guaranteed to be smaller than 2^{k+1}; if we ensure that parties always play the OT-receiver when using shares from this set as input, then the concrete bandwidth cost for this second step is $(n-1) \cdot (\kappa+1) \cdot (2\kappa+1) + 2(n-1) \cdot \kappa$ bits transmitted per party, assuming Silent OT is used, or $(n-1) \cdot (\kappa+1) \cdot (2\kappa+\lambda) + 2(n-1) \cdot \kappa$ bits if KOS OT is used. In either case, three additional rounds are required.

Instantiating $\mathcal{F}_{\mathsf{ComCompute}}$. In the malicious settings, the π_{Biprime} and π_{RSAGen} protocols both require access to a generic MPC functionality with reusable inputs, which we give as $\mathcal{F}_{\mathsf{ComCompute}}$. For the sake of concrete-efficiency figures, we will assume $\mathcal{F}_{\mathsf{ComCompute}}$ is realized via the Authenticated Garbling protocol of Wang et al. [WRK17]. We make no optimizations, and in particular, do not replace the OT that they use, even though Silent OT might yield a substantial improvement. Per the authors [Wan20], this protocol has a total cost including preprocessing of

$$\begin{aligned}&(n-1) \cdot (\lambda+s) \cdot (10|C| + |I| + \lambda + 4s) + |I| \cdot (\lambda+1)\\&+ (n-1) \cdot (7|C| \cdot \lambda + 13|C| + 7\lambda) + 4|C| \cdot s + \lambda + |O|\end{aligned}$$

bits transmitted per party, on average, where C is the number of AND gates, I is the number of input wires, and O is the number of output wires, and where the bucket-size parameter is 3 (our circuits will be large enough to ensure this holds). Since this equation is quite complicated and gives little insight, we report costs associated with $\mathcal{F}_{\mathsf{ComCompute}}$ simply in terms of gates and input/output wires, and convert them into bit costs only when calculating the overall total bandwidth cost for π_{RSAGen}. We note that supplying an input (i.e., calling the `commit` command of $\mathcal{F}_{\mathsf{ComCompute}}$) to the protocol of Wang et al. requires two rounds, and reconstructing an output (i.e., calling the `compute` command) requires one. We assume that all preprocessing can be run concurrently with other tasks, and thus it does not contribute any additional rounds.

Using $\mathcal{F}_{\mathsf{ComCompute}}$ involves converting functions into circuits. The functions we use are easily reduced to a handful of arithmetic operations. We list all nontrivial operations, with costs, in Table 6.7.

Operator	Cost Function	Notes
modadd	$4L + 2$	symmetric; conditional subtraction
intmul1	$2\mathsf{intmul}(L/2) + 13L/2 + 6$ $+ \mathsf{intmul}(L/2 + 1)$	symmetric Karatsuba multiplication
intmul2	$2L_1 \cdot L_2 - L_1 + ({L_2}^2 - L_2)/2$	asymmetric multiplication
modred	$2\mathsf{intmul}(L) + 10L + 13$	Barrett reduction; input length $2L$
modmul1	$\mathsf{intmul}(L) + \mathsf{modred}(L)$	symmetric modular multiplication
modmul2	$5L_1 \cdot L_2 + 2L_2$	asymmetric; conditional subtraction

Table 6.7: Operation Costs in AND Gates. For *symmetric* operations, both inputs and (if relevant) the modulus are of length L. For *asymmetric* operations, the first input and the modulus are of length L_1, and the second input is of length L_2 such that $L_2 \leq L_1$. We assume that moduli are public and that inputs to modular operations are positive and less than the modulus, except in the case of reduction. The intmul operation calls whichever multiplication method is more (concretely) efficient.

$\mathcal{F}_{\mathsf{AugMul}}$ **in the Malicious Setting.** The protocol π_{AugMul} that realizes $\mathcal{F}_{\mathsf{AugMul}}$ is complex and involves a number of sub-functionalities that we have not yet introduced. Consequently, we defer our full cost analysis to Appendix B, in which we also describe π_{AugMul} itself. We note here that the cost of the $\mathcal{F}_{\mathsf{AugMul}}$ `multiply` command with modulus m is in $O(n\cdot(|m|^2+|m|\cdot s+2\lambda))$ transmitted bits per party if Silent OT is used, or $O(n\cdot(|m|^2+|m|\cdot s+(|m|+2)\cdot\lambda))$ if KOS OT is used, and that the `input` command is free if it is not called until immediately before the first associated multiplication. The cost of the `open` command is in $O(n \cdot (|m|^2 + |m| \cdot s))$ transmitted bits per party. The `sample` command has the same expected bandwidth complexity as multiplication. Finally, the complexity of the `check` command depends upon the inputs it is given, but broadly, for each input with an associated modulus m, one must pay $s/|m|$ times the cost of multiplication, plus a bandwidth cost in $O(n \cdot s^2/|m| + n \cdot s \cdot c)$, where c is the number of times the input was used in a multiplication, plus the cost of using $\mathcal{F}_{\mathsf{ComCompute}}$ to evaluate a circuit of size $O(n \cdot s + s \cdot \mathsf{modmul}(|m|)/|m|)$ with $O(s + |m|)$ input wires. In addition to these per input costs, `check` uses $\mathcal{F}_{\mathsf{ComCompute}}$ to evaluate an arbitrary circuit (with no additional inputs), and so costs must be paid proportionate to that circuit's size.

$\mathcal{F}_{\mathsf{Biprime}}$ **in the Malicious Setting.** In Section 5.2, we described a protocol π_{Biprime} realizing $\mathcal{F}_{\mathsf{Biprime}}$. We now analyze its costs. The Boneh-Franklin Test phase is similar to the first phase of the semi-honest biprimality test, except that messages are committed before they are revealed, they are always decommitted concurrently, and $2.5s$ iterations are required, as opposed to s. Consequently this step incurs a cost of $5(n-1)\cdot s\cdot\kappa + 2(n-1)\cdot\lambda$ bits transmitted per party, over two rounds.

This leaves Input Commitment and Consistency Check/GCD Test phases. The former phase involves only a single commitment by each party. The latter phase is *only* evaluated if the above test passes, and therefore does not contribute to the cost when N is not a biprime. To perform the GCD Test, the parties each transmit $2.5s \cdot (n-1) \cdot (3\kappa - \log_2 n + s + 1)$ bits over two rounds, and in addition, they use $\mathcal{F}_{\mathsf{ComCompute}}$ to compute a circuit, which is specified by Algorithm 5.3. If for clarity and simplicity we represent the cost of multiplication using the functions from Table 6.7, then the size of this circuit in AND gates is

$$\begin{aligned}&\mathsf{modmul}(2\kappa) + \mathsf{intmul}(\kappa) + (10n \cdot s + 4n + 2.5s) \cdot (\kappa - \log_2 n)\\&+ 5n \cdot s^2 + 5n \cdot s + 4n + 2\kappa - 2\end{aligned}$$

and furthermore, it has $n \cdot (2.5s + 2) \cdot (\kappa - \log_2 n) + n \cdot 2.5s \cdot (s+1) + 2n \cdot \kappa$ input wires[9] and $2n \cdot \kappa$ output wires.

6.3 Putting It All Together

In this section, we will break down the cost of our main protocol π_{RSAGen} in terms of calls to $\mathcal{F}_{\mathsf{AugMul}}$ and $\mathcal{F}_{\mathsf{Biprime}}$, and also give the costs associated with the few protocol components that are not interactions with these functionalities. This breakdown (excepting the functionality calls in the Consistency Check phase) is identical in the malicious and semi-honest settings. We subsequently plug in the individual component costs that we derived in Section 6.2 and arrive at predicted concrete cost figures for one instance of our complete protocol. We then use the analysis in Section 6.1 to determine the concrete cost of sampling a biprime when our sampling protocol is called *sequentially*. We discuss alternative composition strategies for achieving constant or expected-constant round counts (and provide an exact round count) in the next section. We begin with a cost breakdown.

1. **Candidate Sieving.** For each $j \in [2, \ell]$, the parties invoke the `sample` instruction of $\mathcal{F}_{\mathsf{AugMul}}$ using modulus $\mathbf{m}_j$. These invocations are concurrent. Following this, for $j \in [\ell+1, \ell']$, the parties invoke the `input` instruction of $\mathcal{F}_{\mathsf{AugMul}}$ with modulus $\mathbf{m}_j$ twice, and then the `multiply` instruction with modulus $\mathbf{m}_j$. As discussed in Appendix B.3, the `input` instructions are free under this pattern of invocations if we use π_{AugMul} (Protocol B.6) to realize $\mathcal{F}_{\mathsf{AugMul}}$. We assume the multiplications are done concurrently as well. In addition, every party broadcasts a value in $\mathbb{Z}_{\mathbf{m}_j}$ for $j \in [2, \ell']$. This incurs a cost of
$$(n-1) \cdot \sum_{j \in [2,\ell']} |\mathbf{m}_j|$$
transmitted bits per party, and one additional round.

2. **Biprimality Test.** The parties send `check-biprimality` to $\mathcal{F}_{\mathsf{Biprime}}$ at most once. Since our protocol includes local trial division (see Step 5 of π_{RSAGen}),

[9] Including the wires required to input the randomness for the GCD test.

some protocol instances will skip the biprimality test entirely. However, for the sake of calculating concrete costs, we will assume that this local trial division is not executed. In other words, we assume that the biprimality test is run exactly once per protocol instance, and if local trial division would have rejected a candidate, then the biprimality test certainly fails.

3. **Consistency Check.** This phase has no cost in the semi-honest setting. In the malicious setting, its cost differs depending on whether the candidate biprime has passed the biprimality test (and local trial division), or failed. In the failure case, the parties send `open` to $\mathcal{F}_{\mathsf{AugMul}}$ twice with modulus $\mathbf{m}_j$ for each $j \in [2, \ell']$. In the passing case, they instead send $(\texttt{check}, \textsf{sids}, f)$, where $\textsf{sids}$ is the vector of session IDs associated with values sampled by or loaded into $\mathcal{F}_{\mathsf{AugMul}}$ in the sieving phase, and f is a predicate specified in our protocol. This circuit representation of f comprises $2n$ invocations of $\mathsf{CRTRecon}$ over the *extended* vector $\mathbf{m}$ of small moduli, $2n$ comparisons on inputs of size 2κ, two integer sums, each over n elements of $\kappa - \log_2 n$ bits, one integer multiplication with inputs of length κ, and one equality check over values of length 2κ. We observe that the $\mathsf{CRTRecon}$ algorithm can be implemented in the following way: all intermediate operations can be performed over the integers, with a single modular reduction at the end. Because each intermediate product has one multiplicand that is much shorter than the other, we can use the asymmetric modular multiplication method previously described in Section 6.2. This circuit requires no additional inputs, relative to the ones supplied by π_{AugMul}, and its overall gate count[10] is

$$2n \cdot \left(\begin{array}{l} (\ell' - 1) \cdot (2\kappa + |\max(\mathbf{m})|) + \ell' - \log_2 \ell' \\ + 3\kappa - \log_2 n + \sum\limits_{j \in [\ell']} \mathsf{intmul2}(2\kappa, |\mathbf{m}_j|) + \mathsf{modred}(2\kappa) \end{array} \right) + \mathsf{intmul}(\kappa) - 2$$

Communication Complexity. In order to explore the asymptotic network costs of our protocol in terms of κ, s, and n, we consider λ be in $O(s)$ for the purposes of this analysis. We also assume $s \in O(\kappa) \cap \omega(\log \kappa)$. We give all our asymptotic figures in terms of data transmitted per party.

Combining the cost breakdown in this section with the asymptotic analysis of CRT sampling parameters in Section 6.1, we find that $O(\kappa / \log \kappa)$ invocations of the `sample` and `multiply` interfaces of $\mathcal{F}_{\mathsf{AugMul}}$ are performed during the Candidate Sieving phase of π_{Biprime}. Let $(\mathbf{m}, \ell', \ell, M)$ be the (κ, n)-compatible parameter set of π_{Biprime}, as specified by Definition 4.3. Each `sample` and `multiply` operation involves a modulus from $\mathbf{m}$. Per Lemma 6.2, $\max \mathbf{m} \in O(\log M)$, and we have from Definition 3.4 that $M \in O(2^\kappa)$; thus $\max \mathbf{m} \in O(\kappa)$. Via Section 6.2, this implies that all `multiply` and `sample` operations have a complexity of $O(n \cdot \log^2 \kappa)$ bits transmitted per party in the semi-honest setting, or $O(n \cdot \log^2 \kappa + n \cdot s \cdot \log \kappa)$ in the malicious setting (assuming Silent OT in both

[10] Assuming the intermediate products inside each invocation of $\mathsf{CRTRecon}$ are taken to be of length $2\kappa + |\max(\mathbf{m})|$, and that each party's shares of p and q are taken to be $\kappa - \log_2 n$ bits *after* their lengths are checked.

cases). The total per-party communication cost of the Sieving phase is therefore $O(n \cdot \kappa \cdot \log \kappa)$ in the semi-honest setting, or $O(n \cdot \kappa \cdot s)$ in the malicious setting.

In both the semi-honest and the malicious settings, the communication complexity of a *successful* biprimality test is dominated by the GCD Test. In the semi-honest case, this check comprises a secure multiplication over κ bits, so the cost is in $O(n \cdot \kappa^2)$ bit transmitted per party. In the malicious case, per Section 6.2, this check is performed via a circuit with $O(n \cdot s \cdot \kappa + \kappa^{\log_2 3})$ gates,[11] and per Wang et al. [WRK17], the dominant per-party asymptotic cost of their protocol is in $O(n \cdot s \cdot C / \log C)$, where C is the gate count. Thus the per-party complexity of a successful biprimality test is

$$O\left(\frac{n^2 \cdot s^2 \cdot \kappa + n \cdot s \cdot \kappa^{\log_2 3}}{\log(n \cdot s \cdot \kappa + \kappa^{\log_2 3})}\right)$$

in the malicious case. In both the semi-honest and the malicious settings, *failed* biprimality tests are unlikely to perform the GCD check,[12] leaving only the initial testing phase. If we choose the bandwidth-optimal protocol variant in the semi-honest setting, then we can take the communication complexity of a failed biprimality test to be in $O(n \cdot \kappa)$. In the malicious setting, it is in $O(n \cdot s \cdot \kappa)$.

Finally, in the malicious setting only, the Consistency Check phase is performed. For presumptive biprimes, the cost of this phase is dominated by its generic MPC component, which has $O(n \cdot \kappa^2)$ gates and, if we assume that Wang et al.'s protocol is used, a total complexity of

$$O\left(\frac{n^2 \cdot s \cdot \kappa^2}{\log n + \log \kappa}\right)$$

per party. For presumptive non-biprimes, the parties invoke the `open` interface of $\mathcal{F}_{\mathsf{AugMul}}$ a constant number of times for each element of $\mathbf{m}$. Reusing our analysis of the sieving phase, we find that the per-party network complexity of this case is $O(n \cdot s \cdot \kappa)$.

In the case that a biprime is sampled by iterating π_{RSAGen} sequentially until a successful sampling occurs, we know (via Section 6.1) that $O(\kappa^2 / \log^2 \kappa)$ iterations are required in expectation. All but one of these iterations are failures, and in the one successful case in the malicious setting, the size of the consistency check circuit dominates that of the biprimality test circuit. Thus, the overall per-party communication complexity to sample a biprime using this composition strategy is in

$$O\left(\frac{n \cdot \kappa^3}{\log \kappa}\right) \qquad \text{or} \qquad O\left(\frac{n \cdot s \cdot \kappa^3}{\log^2 \kappa} + \frac{n^2 \cdot s \cdot \kappa^2}{\log n + \log \kappa}\right)$$

in the semi-honest or malicious settings, respectively.

[11] $O(\kappa^{\log_2 3})$ is the cost of Karatsuba multiplication on κ-bit inputs.

[12] We do not know the precise probability, nor does any prior work, including that of Boneh and Franklin [BF01] themselves, make a statement about it. It seems empirically that the probability is very low indeed. For analysis purposes we take it to be zero.

κ	1024	1024	1536	1536	2048	2048
n	2	16	2	16	2	16
per-phase costs for one instance of π_{RSAGen}						
Sieving (S)	24	365	38	566	52	787
Sieving (K)	302	4529	449	6737	603	9038
BP Test (N)	4	61	6	92	8	122
BP Test (BS)	2266	33,991	4972	74,580	8726	130,897
BP Test (BK)	2396	35,944	5167	77,508	8986	134,801
expected costs to sample a biprime via sequential iteration of π_{RSAGen}						
E[Iterations]	3607	3607	7251	7251	11,832	11,832
E[Total] (S)	104,755	1,571,324	322,980	4,844,704	726,223	10,893,340
E[Total] (K)	1,105,983	16,589,742	3,306,014	49,590,209	7,234,834	108,522,514

Table 6.8: Communication Per Party in Kilobits: Semi-Honest. Recall that biprimes produced are of size 2κ. Values are rounded to the nearest 1 Kilobit, and $s = 80$ and $\lambda = 128$ for all calculations. Rows marked with S or K correspond to costs when Silent OT [BCG+19] or KOS OT [KOS15] is used to instantiate oblivious transfer, respectively. Rows marked with B or N correspond to costs when a Biprime or a Non-biprime is sampled, respectively.

Concrete Network Cost. We now perform all the substitutions necessary to collapse the cost figures we have reported into concrete values. We plug the costs of individual sub-protocols from Section 6.2 into the main protocol breakdown presented in this section, using bandwidth-optimal variants where appropriate, and convert gate counts into bit counts assuming the protocol of Wang et al. [WRK17]. We also derive the expected cost of sampling a single biprime via sequential iteration of π_{RSAGen} until a success occurs: this strategy allows us to determine the number of iterations required by inverting the success probabilities calculated in Section 6.1. Finally, we set $\lambda = 128$ and $s = 80$. We report the results of this calculation for our semi-honest protocol variant in Table 6.8, and for our malicious protocol variant in Table 6.9.

6.4 Strictly-Constant and Expected-Constant Rounds

In the foregoing sections we have given costs for sampling a biprime under the assumption that π_{RSAGen} is iterated *sequentially*. While this ensures that no work or communication is wasted, since no additional instances of π_{RSAGen} are run after the first success, it also yields an expected round complexity that grows with $\Omega(\kappa^2/\log^2\kappa)$ at the least (the inverse of the success probability of a single instance, per Section 6.1), and concretely yields tens or hundreds of thousands of rounds in expectation for the values of κ used in practice. This is not satisfying.

Unfortunately, we cannot achieve constant (or even expected constant) rounds by running instances of π_{RSAGen} concurrently, because the `sample` op-

κ	1024	1024	1536	1536	2048	2048
n	2	16	2	16	2	16
per-phase costs for one instance of π_{RSAGen}						
Sieving (S)	0.7	10.8	1.1	15.8	1.4	21.0
Sieving (K)	6.4	96.6	9.0	135.4	11.5	172.5
BP Test (N)	0.4	6.1	0.6	9.2	0.8	12.3
BP Test (B)	28,383.2	935,581.6	51,549.4	1,516,411.0	78,872.5	2,153,965.0
Check (N)	0.8	12.3	1.2	18.2	1.6	24.3
Check (B)	182,280.6	19,239,118.0	381,065.5	40,509,208.9	652,491.1	69,666,149.6
expected costs to sample a biprime via sequential iteration of π_{RSAGen}						
E[Iterations]	3607	3607	7251	7251	11,832	11,832
E[Total] (S)	217,690.8	20,280,103.8	453,505.1	42,338,971.8	776,737.9	72,500,729.2
E[Total] (K)	238,328.8	20,589,674.7	511,297.6	43,205,860.3	896,225.8	74,293,046.9

Table 6.9: Communication Per Party in Megabits: Malicious. Recall that biprimes produced are of size 2κ. Values are rounded to the nearest 100 Kilobits, and $s = 80$ and $\lambda = 128$ for all calculations. Rows marked with S or K correspond to costs when Silent OT [BCG+19] or KOS OT [KOS15] is used to instantiate oblivious transfer, respectively. Rows marked with B or N correspond to costs when a Biprime or a Non-biprime is sampled, respectively. Note that the Biprimality Test phase has nearly the same concrete cost, regardless of whether Silent OT or KOS OT is used.

eration of π_{AugMul} (which samples pairs of secret-shared non-zero values) is also probabilistic, with termination in expected-constant rounds. When multiple expected-constant-round protocols are composed in parallel, the resulting protocol does not have constant rounds in expectation [BE03, CCGZ19, CCGZ17]. The naïve solution is to modify the `sample` operation of π_{AugMul} to generate candidates concurrently instead of sequentially, and set the parameters such that it succeeds with overwhelming probability. However, this leads to a factor of $\Theta(s)$ overhead with respect to communication complexity, relative to sequential composition, which is also not satisfying.

Instead, we propose a non-black-box composition strategy. In order to invoke π_{RSAGen} x times concurrently, we must sample x pairs of non-zero values with respect to each modulus in $\mathbf{m}$. In order to do this, we run a pooled sampling procedure for each modulus, which concurrently samples enough candidates to ensure that there are at least x successes among them with overwhelming probability. For modulus $\mathbf{m}_j$, and candidate count $y \geq x$, the probability of success is governed by the binomial distribution on y and $(\mathbf{m}_j - 1)^3/\mathbf{m}_j^3$. Specifically, to ensure success with overwhelming probability, we must calculate

y such that

$$1 - 2^{-s} = \sum_{i=x}^{y} \binom{y}{x} \left(\frac{\mathbf{m}_j - 1}{\mathbf{m}_j}\right)^{3i} \cdot \left(1 - \left(\frac{\mathbf{m}_j - 1}{\mathbf{m}_j}\right)^{3}\right)^{y-i} \tag{5}$$

holds. It is not easy to solve this equation for y in closed form,[13] but given concrete values for the other parameters, we can compute a value for y.

As an example, consider the following concrete scenario: let x be the size of a batch of concurrent invocations of π_{RSAGen}, where x is tuned such that the batch samples at least one biprime with probability $1/2$. Let $\kappa = 1024$ and let $s = 80$, and via Section 6.1, we have $x = 2500$. Now consider $\mathbf{m}_2 = 3$, the smallest modulus with respect to which sampling will occur. Solving Equation (5), we find that if we sample 9989 candidate values modulo $\mathbf{m}_2$, then at least 2500 of them will be nonzero with probability $1 - 2^{-s}$. Now, if we sample a biprime by running batches of size x, one batch after the next, until a biprime is sampled, then two batches are required in expectation, which implies that 19978 total candidate pairs must be sampled with respect to $\mathbf{m}_2$, in expectation.

For comparison, consider the sequentially composed case with sequential sampling, in which single instances of π_{RSAGen} are invoked one-at-a-time until a success occurs, and similarly π_{AugMul} samples candidate pairs of non-zero values until a success occurs. In this case, 3607 invocations of π_{RSAGen} are required in expectation, and during each invocation, $(3/2)^3$ candidate pairs are sampled in expectation by π_{AugMul} with respect to the modulus $\mathbf{m}_2 = 3$. The total number of candidate pairs with respect to $\mathbf{m}_2$ is thus 12174 in expectation over all invocations of π_{RSAGen}. Thus, in terms of candidates-with-respect-to-$\mathbf{m}_2$, the overhead to achieve expected constant rounds using the above strategy is a factor of roughly 1.65, and it is easy to confirm that for all other elements in $\mathbf{m}$, this factor is smaller. We conclude from this example that the bandwidth overhead for sampling a biprime in expected-constant rounds is reasonable, relative to the bandwidth-optimal (i.e., sequential) composition strategy. By adjusting the batch-size parameter, it can be shown that there exists a biprime-sampling strategy that takes strict-constant rounds and succeeds in sampling a biprime with probability $1-2^{-40}$, and that the bandwidth overhead of this strategy is a factor of less than 30 with respect to the bandwidth-optimal strategy (assuming, as before, that $\kappa = 1024$ and $s = 80$).

An Exact Round Count. It remains only to determine the exact round count of a batch, which is the same as the round count of a single instance of π_{RSAGen}, assuming the sampling performed by π_{AugMul} in that instance is performed concurrently, and that the round-optimal variants of the biprimality test are used. In the semi-honest setting, if we modify the simple semi-honest realization of $\mathcal{F}_{\mathsf{AugMul}}$ given in Section 6.2 to sample concurrently instead of sequentially then it requires exactly 5 rounds to sample. A similar modification yields 10 rounds for sampling using π_{AugMul} in the malicious setting. All other

[13] Although reasonable bounds on and approximations for y exist.

components of our combined protocol have constant round counts, and combining the round counts already given in Section 6.2, Section 6.3, and Appendix B yields a total of 12 rounds in the semi honest setting, or 33 rounds in the malicious setting. It is likely that both values can be reduced noticeably via concrete optimization. If the parties are interacting for the first time, they will need a small number of additional rounds to initialize OT-extensions.

6.5 Comparison to Prior Work

The most relevant prior work is that of Frederiksen et al. [FLOP18], and in this section we provide an in-depth comparison of the two. Unfortunately, Frederiksen et al. do not report concrete communication costs, and so we must re-derive them as best as we can. Before we discuss concrete costs, however, we describe the high-level differences between the two approaches, and our methodology for comparison. As their protocol supports only two parties, we will assume for the duration of this section that $n = 2$.

Both their protocol and ours are based upon oblivious transfer, but the sampling mechanisms of the two protocols have substantial structural differences. Our CRT-form sampling mechanism has a complexity in $O(\kappa \cdot \log \kappa)$ or $O(\kappa \cdot s)$ in the semi-honest or malicious settings, respectively. This complexity includes the cost of computing the modulus N from shares of its factors. Their protocol instead samples integer shares of p and q uniformly in *standard* form and uses 1-of-many OT [OOS17] to perform trial division efficiently. Subsequently, they use an OT-multiplier of their own design to compute $N \coloneqq p \cdot q$. Because this multiplication is not performed in CRT form, it has a communication complexity of $O(\kappa^2)$ in both the semi-honest and the malicious settings. In addition, their method induces some leakage, whereas ours leaks no more than Boneh and Franklin [BF01]. When a biprime is sampled successfully, their biprimality test uses another, similar multiplier, with inputs that are larger by a constant factor. Since their protocol makes black-box use of the KOS OT-extension protocol [KOS15], our comparison will assume their protocol is upgraded to use the more-recent Silent OT-extension protocol [BCG$^+$19]. Furthermore, we will assume their scheme amortizes the costs of both 1-of-2 and 1-of-many OT-extension in the best possible way (i.e., we will only count costs that can never be amortized).

In the malicious setting, Frederiksen et al.'s protocol makes use of a *proof of honesty* which consists mainly of a circuit evaluated by a generic MPC functionality. It is quite similar to a combination of the circuits used by our π_{Biprime} protocol and the Consistency Check phase of our π_{RSAGen} protocol.[14] However, they must run their circuit twice, and on the other hand, because their protocol is two-party only and already admits leakage, they are able to use the very efficient leaky dual-execution method of Mohassel and Franklin [MF06] to evaluate

[14] As described in their work, their circuit contains additional gates for calculating parts of the RSA key-pair other than the modulus. We omit these parts from our analysis, for the sake of fairness.

their circuit, whereas we must use an n-party method with full malicious security. Our goal is to illustrate performance differences between our approach and theirs, not performance differences among underlying black-box components. Because we do not think there is a fair way to reconcile the discrepancy in the generic MPC techniques that the two works employ, we will report the number of AND gates required by both protocols,[15] and also the number of bits transmitted per party *excluding* those due to evaluation of these circuits.

Finally, as Frederiksen et al. observed and as we have previously discussed, their functionality allows a malicious adversary to covertly prevent successful sampling in a selective fashion. This is true regardless of the composition strategy employed. As a consequence, in the malicious setting, we will report both the best-case behavior, in which the adversary politely avoids covert cheating, and the worst-case behavior, in which they must run sufficiently many iterations to ensure a modulus should have been sampled with overwhelming probability (and thus there must be a corrupt party present if no moduli are produced).

The protocol of Frederiksen et al. takes as a free parameter a trial-division bound, which determines the amount of trial division to be done before the candidate biprime N is reconstructed and biprimality testing occurs. Since they do not report the trial-division bound used for their concrete experiments, nor do they make a recommendation about the value other than that it be tweaked to optimize the protocol cost concretely, we choose their trial-division bound to be the same as the largest element in a $(\kappa, 2)$-near-primorial vector, per Definition 3.4. This is a reasonable choice in terms of concrete efficiency, but it also allows us to easily make a fair comparison, since with this parametrization sampling a biprime will require the same number of invocations of π_{RSAGen} (each with one biprimality test and one consistency check) in our case as there are executions of the biprimality test phase and proof-of-honesty in their case.[16]

The protocol of Frederiksen et al. also takes a second trial division bound, which is used for local trial division on the reconstructed modulus. For the sake of analysis, we assume this feature is not used (we have done likewise with the analogous feature in our own protocol). We calculate concrete costs for their protocol using $s = 80$ and $\lambda = 128$ and report the results alongside our own for the semi-honest setting in Table 6.10, and for the malicious setting in Table 6.11.

In the semi-honest setting, the asymptotic and concrete advantages of our approach are very clear: the overall expected cost advantage of our protocol is a factor of at least 75, and increases with the size of the biprime to be sampled. This advantage is a direct consequence of our CRT-form sampling strategy, since the two semi-honest protocols are otherwise very similar.

In the malicious setting, the relationship between the two protocols is more complex. It is still apparent that for non-successful instances, our protocol is both asymptotically and concretely better. This comes at the price of a larger circuit to be evaluated when an instance succeeds in sampling a biprime. We believe this to be a beneficial trade: only one successful instance is required

[15] We calculate their circuit size using the building blocks we previously described in Table 6.7 and use their estimate of 6000 gates for the cost of a single AES call.

[16] Assuming that in their case, the adversary never forces the rejection of valid candidates.

Scheme	[FLOP18]	Ours	[FLOP18]	Ours	[FLOP18]	Ours
κ	1024	1024	1536	1536	2048	2048
expected costs for one instance/iteration						
E[Kbits] (N)	2263	28	4967	44	8720	61
E[Kbits] (B)	5419	2290	12,061	5010	21,324	8779
expected costs to sample a biprime via sequential iteration						
E[Iterations]	3607	3607	7251	7251	11,832	11,832
E[Total Kbits]	8,165,941	104,755	36,025,885	322,980	103,192,187	726,223

Table 6.10: Comparison against Frederiksen et al: Semi-Honest. Recall that biprimes produced are of size 2κ. We set $s = 80$ and $\lambda = 128$ for all calculations, and all rows assume Silent OT [BCG$^+$19] is used to instantiate 1-of-2 oblivious transfer. Rows marked with B or N correspond to costs when a Biprime or a Non-biprime is sampled, respectively.

Scheme	[FLOP18]	Ours	[FLOP18]	Ours	[FLOP18]	Ours
κ	1024	1024	1536	1536	2048	2048
expected costs for one instance/iteration						
E[Mbits] (N)	4.9	1.9	10.4	2.9	18.1	3.8
E[Mbits] (B)	15.9	72.2	34.1	97.2	59.1	119.6
expected costs to sample a biprime via sequential iteration						
E[Iterations] (P)	3607	3607	7251	7251	11,832	11,832
E[Iterations] (W)	199,968	3607	402,003	7251	656,070	11,832
E[Total Mbits] (P)	17,528.7	7098.4	75,653.1	20,986.3	214,211.9	45,492.6
E[Total Mbits] (W)	971,170.9	7098.4	4,192,998.6	20,986.3	11,875,558.8	45,492.6
Millions of Gates	15.4	63.6	25.3	130.7	36.2	220.9

Table 6.11: Comparison against Frederiksen et al: Malicious. Recall that biprimes produced are of size 2κ, and note that bit counts *exclude* bits due to generic evaluation of circuits, which are counted separately. We set $s = 80$ and $\lambda = 128$ for all calculations, and all rows assume Silent OT [BCG$^+$19] is used to instantiate 1-of-2 oblivious transfer. Rows marked with B or N correspond to costs when a Biprime or a Non-biprime is sampled, respectively. Rows marked with W or P correspond to costs when a Worst-case or a Polite (i.e., best-case) adversary is present, respectively.

to sample a biprime, whereas thousands of unsuccessful instances are to be expected in the best case. However, the most noticeable difference between the two arises due to the ability of a malicious adversary to covertly force instances of the Frederiksen et al. protocol to fail. This can cause the number of instances they require to be inflated by a factor of around fifty, assuming sequential iteration, giving our approach a clear advantage.

Acknowledgements

The authors thank Muthuramakrishnan Venkitasubramaniam for the useful conversations and insights he provided, Tore Frederiksen for reviewing and confirming our cost analysis of his protocol [FLOP18], and Xiao Wang and Peter Scholl for providing detailed cost analyses of their respective protocols [WRK17, HSS17].

This research was supported in part by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Project Activity (IARPA) under contract number 2019-19-020700009 (ACHILLES).

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of ODNI, IARPA, DoI/NBC, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

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.

[Bar16] Elaine Barker. Nist special publication 800-57, part 1, revision 4. `http://dx.doi.org/10.6028/NIST.SP.800-57pt1r4`, 2016.

[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.

[BE03] Michael Ben-Or and Ran El-Yaniv. Resilient-optimal interactive consistency in constant time. *Distributed Computing*, 16(4):249–262, 2003.

[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.

[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.

[CCGZ17] Ran Cohen, Sandro Coretti, Juan Garay, and Vassilis Zikas. Round-preserving parallel composition of probabilistic-termination cryptographic protocols. In *Proceedings of the 44th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 37:1–37:15, 2017.

[CCGZ19] Ran Cohen, Sandro Coretti, Juan A. Garay, and Vassilis Zikas. Probabilistic termination and composability of cryptographic protocols. *Journal of Cryptology*, 32(3):690–741, 2019.

[CDI05] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Proceedings of the Second Theory of Cryptography Conference, TCC 2005*, pages 342–362, 2005.

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

[CL17] Ran Cohen and Yehuda Lindell. Fairness versus guaranteed output delivery in secure multiparty computation. *Journal of Cryptology*, 30(4):1157–1186, 2017.

[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.

[EGL85] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Communications of the ACM*, 28(6):637–647, 1985.

[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.

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

[GL05] Shafi Goldwasser and Yehuda Lindell. Secure multi-party computation without agreement. *Journal of Cryptology*, 18(3):247–287, 2005.

[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.

[IOZ14] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multi-party computation with identifiable abort. In *Advances in Cryptology – CRYPTO 2014, part II*, pages 369–386, 2014.

[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.

[MF06] Payman Mohassel and Matthew K. Franklin. Efficiency tradeoffs for malicious two-party computation. In *Proceedings of the 9th International Conference on the Theory and Practice of Public-Key Cryptography (PKC)*, pages 458–473, 2006.

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

[MWB99] Michael Malkin, Thomas Wu, and Dan Boneh. Experimenting with shared RSA key generation. In *Proceedings of the Internet Society's 1999 Symposium on Network and Distributed System Security*, pages 43–56, 1999.

[OOS17] Michele Orrù, Emmanuela Orsini, and Peter Scholl. Actively secure 1-out-of-n OT extension with application to private set intersection. In *Topics in Cryptology - CT-RSA 2017 - The Cryptographers' Track at the RSA Conference*, pages 381–396, 2017.

[Pie19] Krzysztof Pietrzak. Simple verifiable delay functions. In *Proceedings of the 10th Annual Innovations in Theoretical Computer Science (ITCS) conference*, pages 60:1–60:15, 2019.

[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.

[Wan20] Xiao Wang. Private Communication, 2020.

[Wes19] Benjamin Wesolowski. Efficient verifiable delay functions. In *Advances in Cryptology – EUROCRYPT 2019, part III*, pages 379–407, 2019.

[WRK17] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *Proceedings of the 24th ACM Conference on Computer and Communications Security, (CCS)*, pages 39–56, 2017.

A The UC Model and Useful Functionalities

A.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 $\mathcal{P}_1, \ldots, \mathcal{P}_n$ 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 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 $\mathcal{P}_1, \ldots, \mathcal{P}_n$, 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 into believing it is running in the real experiment. The ideal-world experiment completes when $\mathcal{Z}$ stops activating 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 $\mathcal{Z}$

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

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.

A.2 Useful Functionalities

To realize $\mathcal{F}_{\mathsf{AugMul}}$ and $\mathcal{F}_{\mathsf{Biprime}}$, we use a number of 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 an element uniformly at random from an arbitrary domain.

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

This functionality is parametrized 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 Canetti et al. [CLOS02].

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

This functionality is parametrized 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} (Protocol 5.2), we make use of a functionality for randomly sampling integer shares of zero. This functionality can be realized assuming two-party coin tossing via a slight modification of a protocol of Cramer et al. [CDI05].

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

This functionality is parametrized 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 $\mathbf{x} \leftarrow [-B, B]^{n \times n}$ conditioned on $\mathbf{x}_{i,j} + \mathbf{x}_{j,i} = 0$ for all $i \in [n]$ and $j \in [n]$, and send $(\texttt{zero-share}, \mathsf{sid}, \sum_{j \in [n]} \mathbf{x}_{i,j})$ 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 A.4. $\mathcal{F}_{\mathsf{ComCompute}}(n)$**. Commit-and-compute MPC**

This functionality is parametrized 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 OT protocol with a commitment scheme, as we discuss in Appendix B.1. 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-extension [BCG+19] or the KOS OT-extension protocol [KOS15].

Functionality A.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 $\boldsymbol{\alpha} \in \mathbb{G}_1 \times \mathbb{G}_2 \times \ldots \mathbb{G}_{\ell_{\mathsf{OT}}}$ (that is, $\boldsymbol{\alpha}$ is a vector of elements of length ℓ_{OT} 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 sample a random pad $\rho \leftarrow \mathbb{G}_i$, 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 $\mathbb{G}_i$, and store $(\texttt{complete}, \mathsf{sid}, i)$ in memory.

B Instantiating Multiplication

In this section, we describe how to instantiate the $\mathcal{F}_{\mathsf{AugMul}}$ functionality, and discuss the efficiency of our protocols. In Appendix B.1, we begin by using oblivious transfer and commitments to build delayed correlated oblivious transfer ($\mathcal{F}_{\mathsf{DelayedCOT}}$). In Appendix B.2, we use $\mathcal{F}_{\mathsf{DelayedCOT}}$ to realize a two-party multiplier $\mathcal{F}_{\mathsf{ReuseMul2P}}$ that allows inputs to be reused, and postpones the detection of malicious behaviour. In Appendix B.3, we plug this into the classic GMW multiplication technique [GMW87] in order to realize an n-party multiplier $\mathcal{F}_{\mathsf{ReuseMul}}$, with the same properties of input-reuse and delayed cheat detection. Finally, in Appendix B.4, we combine this component with generic multiparty computation ($\mathcal{F}_{\mathsf{ComCompute}}$) via a simple MAC to realize $\mathcal{F}_{\mathsf{AugMul}}$. We discuss security and give concrete efficiency analysis in parallel. In our efficiency analysis, we make the same assumptions and concessions as in Section 6.

B.1 Delayed-Transmission Correlated Oblivious Transfer

Given groups $\mathbb{G}_1, \ldots, \mathbb{G}_{\ell_{\mathsf{OT}}}$, the functionality $\mathcal{F}_{\mathsf{DelayedCOT}}$ with respect to $\mathbb{G}_1 \times \ldots \times \mathbb{G}_{\ell_{\mathsf{OT}}}$ can be realized in the $(\mathcal{F}_{\mathsf{Com}}, \mathcal{F}_{\mathsf{OT}})$-hybrid model, where $\mathcal{F}_{\mathsf{OT}}$ is the standard oblivious-transfer functionality [CLOS02]. Our construction uses a collection of ℓ_{OT} hash functions $H_i : \{0,1\}^\lambda \mapsto \mathbb{G}_i$ such that for $r \leftarrow \{0,1\}^\lambda$, the vector $(H_1(r), \ldots, H_{\ell_{\mathsf{OT}}}(r))$ is indistinguishable from $(g_1, \ldots, g_{\ell_{\mathsf{OT}}}) \leftarrow \mathbb{G}_1 \times \ldots \times \mathbb{G}_{\ell_{\mathsf{OT}}}$. This is trivial when each H_i is modeled as a random oracle.

1. The sender samples two uniformly random messages $(r_0, r_1) \leftarrow \{0,1\}^\lambda \times \{0,1\}^\lambda$, and the receiver (who has an input bit $\beta \in \{0,1\}$) uses $\mathcal{F}_{\mathsf{OT}}$ to receive r_β.

2. In order to commit to sending its input-correlation vector $\boldsymbol{\alpha}$, the sender uses $\mathcal{F}_{\mathsf{Com}}$ to commit to $H_i(r_0) + H_i(r_1) + \boldsymbol{\alpha}_i$ for $i \in [|\boldsymbol{\alpha}|]$.

3. To implement the transfer instruction for index i, the sender instructs $\mathcal{F}_{\mathsf{Com}}$

to decommit $x = H_i(r_0) + H_i(r_1) + \boldsymbol{\alpha}_i$, and then the receiver retrieves its output follows:

- If $\beta = 0$, then the receiver can calculate its output $H_i(r_0)$.
- If $\beta = 1$, then the receiver subtracts $H_i(r_1)$ from the value $\mathcal{F}_{\mathsf{Com}}$ delivered to calculate $H_i(r_0) + \boldsymbol{\alpha}_i$.

Meanwhile, the sender outputs $H_i(r_0)$.

It is clear to see that at the end of this protocol, the sender and receiver hold additive shares of $\beta \cdot \boldsymbol{\alpha}_i$ for every index i that has been decommitted, and that the sender is committed to $\boldsymbol{\alpha}$. One may notice a small gap between the protocol and the ideal functionality: the sender's share is available before the transfer phase, as is the receiver's share, if $\beta = 0$. Fixing this is simple but not important for our usage of $\mathcal{F}_{\mathsf{DelayedCOT}}$, and so we omit it.

Theorem B.1. *Let $\mathbb{G}_1, \ldots, \mathbb{G}_{\ell_{\mathsf{OT}}}$ be groups and assume there exists a collection $\{H_i : \{0,1\}^\lambda \mapsto \mathbb{G}_i\}_{i \in [\ell_{\mathsf{OT}}]}$ as described above. Then, there exists a protocol in the $(\mathcal{F}_{\mathsf{OT}}, \mathcal{F}_{\mathsf{Com}})$-hybrid model that UC-realizes $\mathcal{F}_{\mathsf{DelayedCOT}}$ with respect to $\mathbb{G}_1 \times \ldots \times \mathbb{G}_{\ell_{\mathsf{OT}}}$, which requires a single invocation of $\mathcal{F}_{\mathsf{OT}}$.*

We note that with a particular usage pattern, there is a more efficient instantiation available, based upon a non-black-box usage of either Silent OT [BCG$^+$19] or KOS OT [KOS15], which we introduced in Section 6.2. In particular, assume there is some specific point in time, and that all correlations are decommitted either immediately after they are committed, or at that point. Both Silent OT and KOS OT involve sending a message from the receiver to the sender first, followed by a message from the sender to the receiver, where each bit in this latter message corresponds to one bit of the correlation(s) transferred (and any correlation bit can be recovered from only the one associated message bit). Thus, $\mathcal{F}_{\mathsf{DelayedCOT}}$ can be realized in the following context-optimized way: the receiver sends its message, and then the sender sends the bits of its message associated with the correlations to be immediately released, and commits (using a single commitment) to the other bits of its message. These bits are decommitted later. This eliminates the potential overhead associated with transferring the r_0 and r_1 values in our above construction. Furthermore, if many $\mathcal{F}_{\mathsf{DelayedCOT}}$ instances are invoked at once, and the delayed-release correlations are released simultaneously across all instances, then the instances can share a single commitment. Our usage of $\mathcal{F}_{\mathsf{DelayedCOT}}$ matches this pattern, and so its cost is equal (up to a few bits) to that of either Silent OT or KOS OT. When calculating concrete efficiency figures, we assume the overhead is exactly zero.

B.2 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 B.2. $\mathcal{F}_{\mathsf{ReuseMul2P}}(m)$**. Two-Party Multiplication**

This functionality is parametrized 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 $0 \le x < m$, 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 $0 \le z_c < m$,[a] if sid_3 has not previously been used, 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 $0 \le z_h < m$. 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 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.

Theorem B.3. *The functionality $\mathcal{F}_{\mathsf{ReuseMul2P}}$ can be UC-realized in the $\mathcal{F}_{\mathsf{DelayedCOT}}$ hybrid model.*

The proof of this theorem is via construction of a protocol $\pi_{\mathsf{ReuseMul2P}}$, which we sketch here, along with a security argument. We adapt the multiplication protocols of Doerner et al, and refer the reader to their work [DKLS19, Section 3] for a more in-depth technical explanation.

Common parameters and hybrid functionalities. The protocol $\pi_{\mathsf{ReuseMul2P}}$ is parametrized by the statistical security parameter s and a prime $m \in O(\log s)$. For convenience, we define a *batch-size* $\xi := 2s + |m|$, a *repetition count* $r = \lceil s/|m| \rceil$, and a randomly sampled public *gadget vector* $\mathbf{g} \leftarrow \mathbb{Z}_m^{\xi}$. Looking ahead, Bob will encode his input via an inner product of ξ random bits and the gadget vector $\mathbf{g}$. The parameter ξ is set so that deleting up to s elements of $\mathbf{g}$ still leaves sufficient rank in $\mathbf{g}$ to encode any element $\mathbb{Z}_m$ with overwhelming probability. The participating parties have access to the commitment functionality $\mathcal{F}_{\mathsf{Com}}$ and the delayed-transfer COT functionality $\mathcal{F}_{\mathsf{DelayedCOT}}$.

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 $\textsf{bit-sids} := \{\mathsf{GenSID}(\mathsf{sid}, j)\}_{j \in [\xi]}$.
2. Alice samples a consistency-check vector $\tilde{\mathbf{a}} \leftarrow \mathbb{Z}_m^r$.
3. Bob samples a vector of choice bits $\boldsymbol{\beta} \leftarrow \{0,1\}^{\xi}$.
4. For each $i \in [\xi]$ (concurrently):
   (a) Bob sends $(\texttt{choose}, \textsf{bit-sids}_i, \boldsymbol{\beta}_i)$ to $\mathcal{F}_{\mathsf{DelayedCOT}}$, and Alice is notified.
   (b) Alice sends $(\texttt{commit}, \textsf{bit-sids}_i, (a, \tilde{\mathbf{a}}_1, \ldots, \tilde{\mathbf{a}}_r) \in \mathbb{Z}_m^{r+1})$ 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. [DKLS18, DKLS19], 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 ensure 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 $b \coloneqq \delta + \langle \mathbf{g}, \boldsymbol{\beta} \rangle \bmod m$. 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 $\delta \leftarrow \mathbb{Z}_m$ and sends it to Alice. Doerner et al. 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 $\xi = 2s + |m|$ invocations of $\mathcal{F}_{\mathsf{DelayedCOT}}$ per multiplication, each invocation with a correlation of size $|m|$. If we realize $\mathcal{F}_{\mathsf{DelayedCOT}}$ via Silent OT, then Alice must transmit $|m| \cdot (|m| + 2s) + 2\lambda$ bits in total and Bob must transmit $2|m| + 2s$ bits it total. If we realize $\mathcal{F}_{\mathsf{DelayedCOT}}$ via KOS OT, then Alice must transmit $|m| \cdot (|m| + 2s) + 2\lambda$ bits in total and Bob must transmit $\lambda \cdot (|m| + 2s) + |m|$ bits in total. Regardless, they require three rounds.

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 Predicate 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 $\textsf{sids}$ of all the multiplications in which Alice was expected to use this input, along with a vector $\textsf{bit-sids}$ of the individual $\mathcal{F}_{\mathsf{DelayedCOT}}$ instances (over all of the multiplications associated with $\textsf{sids}$) 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 $\textsf{bit-sids}$. 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 $i \in [|\textsf{bit-sids}|]$ 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 $0 \leq \tilde{\mathbf{z}}_{\mathsf{B},i,j} < m$. Note that if Alice has behaved honestly, then per the specification of $\mathcal{F}_{\mathsf{DelayedCOT}}$, it holds for all $i \in [|\textsf{bit-sids}|]$ and $j \in [r]$ that
$$\tilde{\mathbf{z}}_{\mathsf{A},i,j} + \tilde{\mathbf{z}}_{\mathsf{B},i,j} \equiv \boldsymbol{\beta}_i \cdot \tilde{\mathbf{a}}_j \pmod{m}$$

2. Bob sends a random challenge $\mathbf{e} \leftarrow \mathbb{Z}_m^r$ to Alice.

3. Alice computes her responses
$$\begin{aligned}\boldsymbol{\psi} &:= \{(\tilde{\mathbf{a}}_j + \mathbf{e}_j \cdot a) \bmod m\}_{j \in [r]} \\ \boldsymbol{\zeta} &:= \{(\tilde{\mathbf{z}}_{\mathsf{A},i,j} + \mathbf{e}_j \cdot \mathbf{z}_{\mathsf{A},i}) \bmod m\}_{i \in [|\textsf{bit-sids}|],\, j \in [r]}\end{aligned}$$

4. Bob verifies that for each $i \in [|\textsf{bit-sids}|]$ 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. [DKLS18] proved 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 or Silent OT 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 $\textsf{bit-sids}$ 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 $0 \le \tilde{\mathbf{z}}_{\mathsf{A},i,j} < m$, while Bob receives $\tilde{\mathbf{z}}_{\mathsf{B},i,j}$ such that $0 \le \tilde{\mathbf{z}}_{\mathsf{B},i,j} < m$.

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 $\textsf{\textbf{bit-sids}}$, Alice decommits $(a, \{\mathbf{z}_{\mathsf{A},i}\}_{i\in[|\textsf{\textbf{bit-sids}}|]})$ via $\mathcal{F}_{\mathsf{Com}}$. Bob verifies that for each $i \in [|\textsf{\textbf{bit-sids}}|]$, 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 $\boldsymbol{\beta} \in \{0,1\}^{\xi}$ 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, $|m| \cdot (|m| + 2s)$ bits for Alice, and 3 messages in total.

B.3 Multiparty Reusable-Input Multiplier

Plugging $\mathcal{F}_{\mathsf{ReuseMul2P}}$ into a GMW-style multiplication protocol [GMW87] 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, by Doerner et al. [DKLS19]), and the security argument follows along similar lines to prior work. 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 B.4. $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$**. Multiparty Multiplication**

This functionality is parametrized 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 $0 \leq x_i < m$ 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 $0 \leq z_i < m$ 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.

B.4 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.

Theorem B.5. *The protocol π_{AugMul} UC-realizes $\mathcal{F}_{\mathsf{AugMul}}$ in the ($\mathcal{F}_{\mathsf{ReuseMul}}$,$\mathcal{F}_{\mathsf{ComCompute}}$)-hybrid model.*

Protocol B.6. $\pi_{\mathsf{AugMul}}(n)$. Augmented Multiplication

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

Sampling: Upon receiving $(\texttt{sample}, \mathsf{sid}_1, \mathsf{sid}_2, m)$ from the environment where sid_1 and sid_2 are fresh values, each party $\mathcal{P}_i$ for $i \in [n]$ sets $\mathsf{ctr} := 0$ and $\mathsf{sid}_x := \mathsf{GenSID}(\mathsf{sid}_1, \mathsf{sid}_2, x)$ for $x \in [3, 6]$ and then:

1. For every $x \in [1, 6]$, $\mathcal{P}_i$ computes $\mathsf{ctrsid}_x := \mathsf{GenSID}(\mathsf{sid}_x, \mathsf{ctr})$.
2. $\mathcal{P}_i$ 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 $(\texttt{input}, \mathsf{ctrsid}_1, x_i)$, $(\texttt{input}, \mathsf{ctrsid}_2, y_i)$, and $(\texttt{input}, \mathsf{ctrsid}_3, r_i)$, waiting after each for confirmation.
3. $\mathcal{P}_i$ sends $(\texttt{multiply}, \mathsf{ctrsid}_1, \mathsf{ctrsid}_2, \mathsf{ctrsid}_4)$ to $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$, and receives $(\texttt{product}, \mathsf{ctrsid}_4, z_i)$ in response.
4. $\mathcal{P}_i$ sends $(\texttt{input}, \mathsf{ctrsid}_5, z_i)$ to $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$, and waits for confirmation, after which it sends $(\texttt{multiply}, \mathsf{ctrsid}_3, \mathsf{ctrsid}_5, \mathsf{ctrsid}_6)$ to $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$, and receives $(\texttt{product}, \mathsf{ctrsid}_6, \tilde{z}_i)$ in response.
5. $\mathcal{P}_i$ 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 $(\texttt{sample}, \mathsf{sid}_1, \mathsf{ctr}, x_i, m)$ and $(\texttt{sample}, \mathsf{sid}_2, \mathsf{ctr}, y_i, m)$ in memory and outputs $(\texttt{sampled-product}, \mathsf{sid}_1, \mathsf{sid}_2, x_i, y_i, z_i)$ to the environment if and only if

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

Otherwise, $\mathcal{P}_i$ sends $(\texttt{open}, \mathsf{ctrsid}_1)$, $(\texttt{open}, \mathsf{ctrsid}_2)$, and $(\texttt{open}, \mathsf{ctrsid}_3)$ 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

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

then $\mathcal{P}_i$ aborts. Otherwise, $\mathcal{P}_i$ increments ctr and begins again from Step 1.

Input: Each party $\mathcal{P}_i$ for $i \in [n]$ begins this protocol phase upon receiving $(\texttt{input}, \mathsf{sid}, x_i, m)$ from the environment where $0 \leq x_i < m$ and sid is a fresh value.

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

7. $\mathcal{P}_i$ stores $(\texttt{value}, \mathsf{sid}, 1, x_i, m)$ in memory.

Multiplication: Each party $\mathcal{P}_i$ for $i \in [n]$ begins this protocol phase upon receiving $(\texttt{multiply}, \mathsf{sid}_1, \mathsf{sid}_2, \mathsf{sid}_3)$ from the environment, where 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 (either or both of these records may also be of type $\texttt{sample}$), and where sid_3 is a fresh value.

8. $\mathcal{P}_i$ sends $(\texttt{multiply}, \mathsf{GenSID}(\mathsf{sid}_1, \mathsf{ctr}_1), \mathsf{GenSID}(\mathsf{sid}_2, \mathsf{ctr}_2), \mathsf{sid}_3)$ to $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$, and receives $(\texttt{product}, \mathsf{sid}_3, z_i)$ in response, which it forwards to the environment.

Predicate Cheater Check: Each party $\mathcal{P}_i$ for $i \in [n]$ begins this protocol phase upon receiving $(\texttt{check}, \textbf{sids}, f)$ from the environment, where $\textbf{sids}$ is a vector of input session IDs such that for each $\mathsf{sid} \in \textbf{sids}$, there exists a record of the form $(\texttt{value}, \mathsf{sid}, *, *, *)$ or $(\texttt{sample}, \mathsf{sid}, *, *, *)$ in $\mathcal{P}_i$'s memory, and where f is the description of a predicate over the stored values associated with the input session IDs. $\mathcal{P}_i$ does nothing if a previous iteration of this protocol phase or the Input Revelation phase included any of the session IDs in $\textbf{sids}$. Let s be the statistical security parameter, and for convenience, let $\mathsf{joint\text{-}sid} := \mathsf{GenSID}(\textbf{sids})$, and let $\mathbf{m}$ be a vector (without duplication) of all the moduli associated with the records referenced by the IDs in $\textbf{sids}$, and let $\mathsf{filter}(\textbf{sids}, m)$ denote the subvector of IDs in $\textbf{sids}$ associated with records that have the modulus m.

9. For each $m \in \mathbf{m}$ (concurrently):

(a) For each $\mathsf{sid} \in \mathsf{filter}(\mathsf{\textbf{sids}}, m)$, each party $\mathcal{P}_i$ for $i \in [n]$ retrieves its record $(\texttt{value}, \mathsf{sid}, *, x_i, m)$ (or $\texttt{sample}$) from memory and sends $(\texttt{commit}, \mathsf{GenSID}(\mathsf{joint\text{-}sid}, 1, i, \mathsf{sid}), x_i)$ 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 $(\texttt{commit}, \mathsf{GenSID}(\mathsf{joint\text{-}sid}, 2, i, j, m), \mathbf{r}_{i,j})$ to $\mathcal{F}_{\mathsf{ComCompute}}(n)$ and $(\texttt{input}, \mathsf{GenSID}(\mathsf{joint\text{-}sid}, 2, j), \mathbf{r}_{i,j})$ 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 $(\texttt{value}, \mathsf{sid}, \mathsf{ctr}, x_i, m)$ (or $\texttt{sample}$) and sends $(\texttt{multiply}, \mathsf{GenSID}(\mathsf{sid}, \mathsf{ctr}), \mathsf{GenSID}(\mathsf{joint\text{-}sid}, 2, j), \mathsf{GenSID}(\mathsf{joint\text{-}sid}, 3, j))$ to $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$, and receives $(\texttt{product}, \mathsf{GenSID}(\mathsf{joint\text{-}sid}, 3, j), \mathbf{t}_{i,j})$ in response.

ii. Each $\mathcal{P}_i$ sends $(\texttt{commit}, \mathsf{GenSID}(\mathsf{joint\text{-}sid}, 3, i, j, m), \mathbf{t}_{i,j})$ 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

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

and if $j = 1$ and this sid is associated with a sampled value, then let f' also verify that

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

and let

$$\mathsf{\textbf{check-sids}} := \left\{ \begin{matrix} \mathsf{GenSID}(\mathsf{joint\text{-}sid}, 1, i', \mathsf{sid}), \\ \mathsf{GenSID}(\mathsf{joint\text{-}sid}, 2, i', j, m), \\ \mathsf{GenSID}(\mathsf{joint\text{-}sid}, 3, i', j, m) \end{matrix} \right\}_{i' \in [n]}$$

Each party $\mathcal{P}_i$ sends $(\texttt{compute}, \mathsf{GenSID}(\mathsf{sid}, j), \mathsf{\textbf{check-sids}}, f')$ 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 $(\texttt{check}, \mathsf{GenSID}(\mathsf{joint\text{-}sid}, 2, j))$ to $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$, and aborts if $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$ aborts.

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

10. Let
$$\mathbf{pred\text{-}sids} := \{\mathsf{GenSID}(\mathsf{joint\text{-}sid}, 1, i', \mathsf{sid})\}_{i' \in [n], \mathsf{sid} \in \mathbf{sids}}$$
Each party $\mathcal{P}_i$ for $i \in [n]$ sends $(\mathtt{compute}, \mathsf{joint\text{-}sid}, \mathbf{pred\text{-}sids}, f)$ to $\mathcal{F}_{\mathsf{ComCompute}}(n)$ and aborts if $\mathcal{F}_{\mathsf{ComCompute}}(n)$ aborts. Otherwise, $\mathcal{P}_i$ receives the output of the predicate from $\mathcal{F}_{\mathsf{ComCompute}}$, and forwards it to the environment.

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

11. $\mathcal{P}_i$ sends $(\mathtt{open}, \mathsf{GenSID}(\mathsf{sid}, \mathsf{ctr}))$ to $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$, and then waits to receive $(\mathtt{opened\text{-}value}, \mathsf{GenSID}(\mathsf{sid}, \mathsf{ctr}), j, x_j)$ from $\mathcal{F}_{\mathsf{ReuseMul}}(n, m)$ for $j \in [n]$. It then outputs $(\mathtt{opening}, \mathsf{sid}, j, x_j)$ for $j \in [n]$ to the environment.

We now discuss security and efficiency of π_{AugMul}, 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 B.3, 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 $(n-1)\cdot(|m|\cdot(|m|+2s)+2\lambda)$ bits if $\mathcal{F}_{\mathsf{DelayedCOT}}$ is realized via Silent OT [BCG+19] or KOS OT [KOS15]. The last party, $\mathcal{P}_n$, always plays Bob, and pays a cost of $(n-1)\cdot(2|m|+2s)$ bits if $\mathcal{F}_{\mathsf{DelayedCOT}}$ is realized via Silent OT, or $(n-1)\cdot(\lambda\cdot(|m|+2s)+|m|)$ bits if $\mathcal{F}_{\mathsf{DelayedCOT}}$ is realized via KOS OT. The other parties play a mixture of the roles, and thus in general they each pay an average cost[17] of

$$\mathsf{Bits}^{\mathtt{multiply}}_{\mathsf{ReuseMul}}(m) \mapsto (n-1)\cdot\frac{(|m|+1)\cdot(|m|+2s)+|m|+2\lambda}{2}$$

transmitted bits with Silent OT or

$$\mathsf{Bits}^{\mathtt{multiply}}_{\mathsf{ReuseMul}}(m) \mapsto (n-1)\cdot\frac{(|m|+\lambda)\cdot(|m|+2s)+|m|+2\lambda}{2}$$

transmitted bits with KOS OT. Regardless, three rounds are required.

[17] We define a function in order to express other costs in terms of this cost; note that the variables n, s and λ are assumed to be global, and thus for simplicity we do not include them among the function's parameters.

Multiplication. The input phase of π_{AugMul} defers directly to $\mathcal{F}_{\mathsf{ReuseMul}}$. As we have noted in Section B.2, 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 three rounds and an average cost of $2\mathsf{Bits}^{\mathtt{multiply}}_{\mathsf{ReuseMul}}(m)$ transmitted bits per party. 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 (where each invocation opens both parties' inputs). Thus the cost of this phase is

$$\mathsf{Bits}^{\mathtt{open}}_{\mathsf{ReuseMul}}(m) \mapsto (n-1)\cdot(|m|+1)\cdot(|m|+2s)$$

transmitted bits per party on average, and three rounds.

Sampling. This procedure is probabilistic. Specifically, each iteration succeeds with probability $((m-1)/m)^3$. We will analyze the costs associated with iterating sequentially until a value is successfully sampled (as described in π_{AugMul}). So long as only a single instance of the sampling procedure is considered, the expected number of sequential iterations depends only on m, but we note that when multiple instances of the sampling procedure are run concurrently, the expected maximum number of iterations among the concurrent instances grows with the number of instances [CCGZ19]. Such concurrency is required in order to achieve biprime sampling in expected-constant or constant rounds, as discussed in Section 6.4. In order to avoid huge overhead costs, an elaborate analysis is required. We perform this analysis in Section 6.4 and, as we have said, focus here on the sequential case.

In the sequential case, $(m/(m-1))^3$ 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. In addition, every party broadcasts a value in $\mathbb{Z}_m$ to the other parties in each iteration. Thus the average cost per party is

$$\left(\frac{m}{m-1}\right)^3\cdot\left(\begin{matrix}4\cdot\mathsf{Bits}^{\mathtt{multiply}}_{\mathsf{ReuseMul}}(m)+2\cdot\mathsf{Bits}^{\mathtt{open}}_{\mathsf{ReuseMul}}(m)\\ +(n-1)\cdot|m|\end{matrix}\right)-2\cdot\mathsf{Bits}^{\mathtt{open}}_{\mathsf{ReuseMul}}(m)$$

transmitted bits, in expectation, and the expected round count is $10(m/(m-1))^3-3$. For values of m of any significant size, 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 directly to the security of the constituent multipliers.[18] 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 protocol phases, this phase takes an input of flexible dimension and therefore its cost 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 $\lceil s/|m| \rceil$ additional invocations of the $\mathcal{F}_{\mathsf{ReuseMul}}$ Multiplication command, with inputs that have previously been loaded, and then run the Cheater Check command of $\mathcal{F}_{\mathsf{ReuseMul}}$, which implies running the Cheater Check command of $\mathcal{F}_{\mathsf{ReuseMul2P}}$ in a pairwise fashion. Together, these operations incur a cost of

$$\frac{s \cdot \mathsf{Bits}^{\texttt{multiply}}_{\mathsf{ReuseMul}}(m)}{|m|} + (n-1) \cdot s \cdot \left(c + \frac{s}{|m|}\right) \cdot (|m| + 2s)$$

transmitted bits per party, on average. Finally, for every input to be checked, the parties each input $\lceil s/|m|+1 \rceil \cdot |m|$ bits into a generic MPC, and then run a circuit that performs $3 \cdot (n-1) \cdot \lceil s/|m| \rceil$ modular additions and $\lceil s/|m| \rceil$ modular multiplications and equality tests over $\mathbb{Z}_m$. Using the circuit component sizes reported in Section 6.2, the size of this circuit comprises $(3 \cdot (n-1) \cdot \mathsf{modadd}(|m|) + \mathsf{modmul}(|m|) + |m|) \cdot \lceil s/|m| \rceil$ AND gates, with $|m|$ additional gates in the case that the input to be checked was sampled. 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. A handful of additional AND gates are required to combine the results from the predicate and the per-input checks, and the circuit has exactly one output wire.

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 $1/m = 2^{-|m|}$, it is necessary to repeat the process $s/|m|$ times in order to achieve a soundness error of 2^{-s}. 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.

[18] Note that the constituent multipliers in this case admit cheats, which are caught later by the Cheater Check command, if it is invoked