Diogenes: Lightweight Scalable RSA Modulus Generation with a Dishonest Majority

Megan Chen
Northeastern U.
& Ligero Inc.

Carmit Hazay
Bar-Ilan U.
& Ligero Inc.

Yuval Ishai
Technion

Yuriy Kashnikov
Ligero Inc.

Daniele Micciancio
UC San Diego

Tarik Riviere
Ligero Inc.

abhi shelat
Northeastern U.
& Ligero Inc.

Muthu Venkitasubramaniam
U. of Rochester
& Ligero Inc.

Ruihan Wang
Ligero Inc.

Abstract

In this work, we design and implement the first protocol for RSA modulus construction that can support thousands of parties and offers security against an arbitrary number of corrupted parties. In a nutshell, we design the "best" protocol for this scale that is secure against passive corruption, then amplify it to obtain active security using efficient non-interactive zero-knowledge arguments. Our protocol satisfies a stronger security guarantee where a deviating party can be identified when the protocol aborts (referred to as security with identifiable-abort) and allows for "public verifiability".

Our passively secure protocol extends the recent work of Chen et al. (Eprint 2020/370) that, in turn, is based on the blueprint introduced in the original work of Boneh-Franklin protocol (CRYPTO 1997, J. ACM, 2001). Specifically, we reduce the task of sampling a modulus to secure distributed multiplication, which we implement via an efficient threshold additively homomorphic encryption (AHE) scheme based on the Ring-LWE assumption. This results in a protocol where the amortized per-party communication cost grows logarithmically in the number of parties. In order to keep the parties lightweight, we employ an "untrusted" coordinator that is connected to all parties and performs all public and broadcast operations.

We amplify this protocol to obtain active security (with identifiable-abort) by attaching zero-knowledge proofs. We instantiate our ZK proof system by composing two different types of ZK proof systems: (1) the Ligero sub-linear zero-knowledge proof system (Ames et al., CCS 2017), and (2) Σ-protocol for proving the knowledge of a discrete logarithm in unknown order groups (Shoup, Eurocrypt 2000).

We implemented both the passive and the active variants of our protocol and ran experiments using 2 to 4,000 parties. *This is the first such implementation of any MPC protocol that can scale to more than 1,000 parties.* For generating a 2048-bit modulus among 1,000 parties, our passive protocol executed in under 4 minutes and the active variant ran in 22 minutes.

Contents

1 Introduction

We present the first secure computation protocol for sampling a 2048-bit RSA modulus (a product of two secret 1024 bit primes) that can *practically scale to thousands of parties while tolerating a dishonest majority*. Our protocol achieves security against either a passive (i.e., semi-honest) or active (i.e., malicious) adversary, who may corrupt all-but-one of the parties. We implemented and benchmarked the active variant of the protocol. As far as we know, our protocol increases the number of parties that can participate in practical secure computation of this kind by *an order of magnitude*; prior work reports at most 256 parties in a protocol involving a signature task [DKLS19].

The motivation for our work comes from applications of large-scale permissionless consensus. Cryptocurrencies and blockchains have re-invigorated the design of *threshold cryptosystems*, where cryptographic operations under a "secret-key" are delegated across a set of nodes and corrupting up to a given threshold of nodes does not compromise security. The problem of generating a shared RSA modulus, introduced in the seminal work of Boneh and Franklin [BF97, BF01], has recently regained attention owing to new and efficient constructions of so-called *verifiable delay functions* (VDFs) [RSW96, BBBF18, Wes19, Pie19, EFKP19] and zero-knowledge proof systems based on hidden-order groups [BFS19].

Beyond the application to VDFs, distributed RSA key generation is a powerful primitive that is motivated by many old and new applications in threshold cryptography; see [Rab98, Des94, Sho00, CDN01] for some earlier works in this area. One class of applications is generating keys for the public-key encryption scheme of Paillier [Pai99], which is widely used in secure computation protocols. Paillier's encryption is an useful building block because it is additively homomorphic and has short ciphertexts. A recent application that depends on Paillier public-key setup is the reusable non-interactive secure computation protocol from [CDI+19]. Another application of distributed RSA keys is using them as part of the common reference string (CRS) for secure computation in the UC setting [Can01]. This was demonstrated for general functions in [JS07] and for concrete functions such as the Fiat-Shamir authentication protocol [FS86, FFS88], set-intersection [JL09], and oblivious pseudorandom functions [JL09]. As noted above, more recent applications include setting up public parameters for VDFs, which can be used in decentralized systems for generating public random beacons and have many other applications.

1.1 Our Contribution

This paper pushes the boundaries of deployable secure computation protocols for non-trivial tasks. We focus on the RSA modulus generation task because it's useful for a wide range of applications, as discussed above. We were particularly motivated by the immediate application of constructing a concrete VDF implementation that is useful in Ethereum's next generation consensus protocol. This task must be run with $(n-1)$-security, namely security against any subset of the n parties. This contrasts with the easier honest-majority setting, in which simpler techniques based on linear secret sharing can be used. Furthermore, this task needs to accommodate thousands of participating parties in order to ensure the VDF's security.

Contribution overview. Our contributions include improvements to the basic distributed RSA modulus generation algorithm, improvements to the design of secure computation protocols for many parties, and an implementation that involves many system-level optimizations. Specifically, we introduce a new protocol for distributed RSA modulus generation with the following features:

1. Security with *identifiable abort* against an active adversary that corrupts an arbitrary subset of parties.
2. The transcript of the protocol is *publicly-verifiable*.
3. The protocol is concretely efficient and scalable up to 4,000 parties (and beyond).

We now explain the above features in more detail.

Performance. We implemented both the passive and active variants of our protocol and ran experiments ranging from 2 to 4,000 parties geographically distributed across multiple AWS cloud centers. We used `t3.small` instances (2GB RAM, up to 2MBps uplink) for all parties and `r5dn.24xlarge` instances for the (untrusted) coordinator. For generating a 2048-bit modulus among 1,000 parties, our passive variant protocol executed in under 4 minutes on average. For 4,000 parties, our passive protocol executed in 12 minutes. For active security, the 1,000 party case ran in roughly 22 minutes while the 4,000-party instance ran in 76 minutes. Note that our protocol also scales down to the 2-party case, which requires 22 seconds in the passive case and 382 seconds in the active one.

As far as we know, this is the first reported implementation of active security for this problem, as well as the first protocol to claim identifiable-abort security.

Security challenges. An *inherent* challenge of achieving active security is that the adversary can always mount a "denial-of-completion" attack, even by corrupting only a single party [Cle86]. Thus when the number of parties n is large, poor network connections could repeatedly cause the entire protocol to fail. The natural defense against such attacks is to support *identifiable abort*. That is, if the protocol fails to complete, the protocol must (publicly) identify at least one malicious or crashed party. Generally, identifying cheaters is challenging for concretely efficient protocols [IOZ14, SF16, BOS16] since parties must reach a consensus on the cheater's identity.

Another desirable feature of actively secure protocols is *public verifiability*, where an honest party can convince an external third party (e.g., a judge) that another party cheated in the computation but a dishonest party cannot incorrectly accuse an honest party [AO12, BDO14, BOS16]. This property is useful for deterring active attacks by *penalizing* malicious parties.

Our protocol achieves both identifiable abort and public verifiability. An external "auditor" can inspect the protocol's transcript and identify parties who deviated from the protocol's specification. In addition, this auditor can be convinced of the correctness of the protocol's outcome in the sense that every party "knows" an additive share such that the protocol's output is the (valid) RSA composite defined by these shares.

The coordinator model. In contrast to prior works on concretely efficient secure computation, our protocol runs in the *untrusted coordinator model*. As discussed in Appendix 1.3, a natural bottleneck in large-scale secure computation is communication; most practical protocols require $O(|\mathrm{C}| \cdot n^2)$ communication, where $|\mathrm{C}|$ is the size of the computation and n is the number of parties, due to pairwise interaction. To circumvent the pairwise interaction barrier, we employ an *untrusted* third party, called a *coordinator*, with an *aggregate-and-broadcast* functionality. This setup keeps computation and communication lightweight for the parties without compromising any security guarantees. Nevertheless, we note that the coordinator is not necessary. Its actions can be performed by a single party or divided equally among all parties.

Since the coordinator is untrusted, its operations need to be verifiable. In principle, the coordinator can prepare a work-saving proof showing that its actions were honest. However, our experiments show that it is faster to simply store the entire protocol transcript in a cloud service and allow anyone to validate it, eliminating the need for work-saving proofs. This is because the coordinator's computations are performed on public values, in contrast to the parties' (lighter-weight) computations involving secrets.

In summary, we design a protocol that is secure with identifiable abort for any active corruption of the coordinator and up to $n-1$ parties. Since we implement an input-less functionality and the coordinator's actions can be verified via a publicly made available transcript and any deviation caught, it suffices to formally prove security against an adversary to actively corrupts up to $n-1$ parties and passively corrupts the coordinator.

Communication model. We assume an authenticated communication channel between each party and the coordinator resulting in a star topology network. In each round, a single message is transmitted from each

party to the coordinator, followed by a "broadcast" message from the coordinator to all parties.

Architecture. Our system architecture provides the following benefits. First, before the ceremony begins, we deploy a bandwidth throughput test to eliminate parties with slow uplink speeds. Also, if a run aborts due to an incorrect message, the corrupted party is identified and eliminated before another run of the protocol is restarted.

1.2 Overview of Techniques

We begin with the framework implicit in the work of Boneh and Franklin [BF97, BF01]. On a high-level, the protocol proceeds in the following steps:

STEP 1: SAMPLE CANDIDATE PAIRS. Generate candidates $\{(p_i, q_i)\}_{i=1}^{\mathcal{N}}$ for pairs of primes, where the values p_i and q_i are congruent to $3 \bmod 4$ for all i.

STEP 2: PRE-SIEVING. The goal of this step is to reduce the number of samples. In particular, prime candidates that are divisible by small prime factors are eliminated.

STEP 3: GENERATE CANDIDATE COMPOSITE. For every index i that is not eliminated, compute a candidate RSA modulus $N_i = p_i \cdot q_i$. Based on the revealed N_i, further eliminate the candidates that are divisible by prime factors up to some threshold. Let $\mathcal{N}'$ be the set of indices that survive this step.

STEP 4: BI-PRIMALITY TEST. Execute a test to identify a valid modulus among $\{N_i\}_{i \in \mathcal{N}'}$ with better soundness. This is called the biprimality test and requires executing the following two steps for each candidate composite N_i:

1. Sample a random γ such that the Jacobi symbol of γ is 1 over N_i. Eliminate N_i if $\gamma^{(N_i+1-p_i-q_i)/4} \notin \{1, -1\}$. As the error is bounded by $1/2$, this step is repeated s times where s is the statistical security parameter (e.g., 80 or 128).
2. Compute $\mathrm{GCD}(N_i, \phi(N_i))$ and eliminate N_i if the outcome does not equal 1. We note that it is not required to compute $\phi(N_i)$ in order to conclude the GCD outcome. Specifically, it holds that $\mathrm{GCD}(N_i, \phi(N_i)) = \mathrm{GCD}(N_i, N_i - \phi(N_i)) = \mathrm{GCD}(N_i, p_i + q_i - 1)$.

STEP 5: OUTPUT PHASE. Output N_i for every index i that survives the biprimality test.

The main bottleneck in scaling secure computation to a large number of parties is the cost of securely *multiplying* secret values. Recent works [KOS16, ADI+17, HIMV19] rely on efficient realizations of the *oblivious transfer* and *oblivious linear evaluation* primitives to achieve secure multiplication in the two-party setting. However, in the multi-party setting, the communication complexity of these techniques scales *quadratically* with the number of parties. This quadratic overhead is prohibitive in practice when the number of parties is large. Indeed, current implementations for general purpose secure computation [WRK17, KPR18] have not been deployed beyond 128 parties with fast communication links.

Our first design choice is to avoid the quadratic overhead by employing a threshold additively homomorphic encryption scheme ((T)AHE): parties encrypt their shares of each secret and send them to the coordinator, who then aggregates the ciphertexts. At first glance, this seems to lead to circularity, since threshold cryptosystems themselves require an MPC protocol for setup. For example, prior work utilizing the Paillier AHE scheme [CDN01] runs into this issue. To circumvent this problem, we rely on a lattice-based AHE, which has much simpler setup. A distributed coin-tossing protocol suffices to set up the public and private parameters. This leads to *total* communication and computation scaling almost linearly with the number of parties.

Originating from [FH96, CDN01], recent works utilize AHE to realize secure computation for which online cost scales well with the number of parties. However in these protocols, the cost of distributing the setup is much higher than ours, making their setup prohibitively expensive for our motivating application. This also applies to protocols in the SPDZ line of work. They either assume a trusted setup [DPSZ12], are limited to covert security [DKL+13], or scale quadratically with the number of parties [KOS16, KPR18, RST+19].

Our protocol design involves two stages: we start with building a passively secure version then amplify its security using *zero-knowledge proofs*. A major difference from previous works is that we use recent general techniques for lightweight *sublinear* zero-knowledge proofs and compose different proof types to enjoy their respective advantages. Another optimization includes proving correctness only for a single successful protocol iteration and carefully analyzing security in case the adversary cheats in the remaining iterations. As a final practical optimization, we decouple the verification of computations involving public values from computations involving secret values. We use the untrusted coordinator to aggregate the publicly verifiable part of the computation and post it on a bulletin board for any (internal or external) party to verify. This effectively allows us to settle for security against an adversary who can actively corrupt up to $n-1$ parties and passively corrupt the coordinator.

We now give a more technical overview of the two parts of our protocol design.

The passively secure protocol. The main building block of our passively secure protocol is a threshold AHE scheme that performs secure arithmetic operations. At a high level, the protocol is structured as follows. The parties sample additive shares $p_1, \ldots, p_n$ for the first prime p and send their encryptions to the coordinator. The coordinator then combines these encryptions to get a ciphertext c encrypting $p = \sum_i p_i$. Upon receiving c, each party P_i, who holds a share of the second prime q, computes the encryption of $q_i \cdot p$ and sends it to the coordinator. The coordinator aggregates these encryptions, resulting in a ciphertext encrypting $p \cdot q$ that the parties decrypt jointly. In fact, this protocol is semi-malicious secure, meaning that the adversary follows the protocol's instructions honestly while using arbitrary and adaptively chosen random tapes.

Before computing the RSA product with the above protocol, the parties perform a pre-sieving phase that disqualifies candidates that are divisible by the first 150 small primes. Our protocol uses the technique from [CCD+20] based on the Chinese Remainder Theorem, which samples non-zero residues modulo small primes to ensure that the reconstruction is not divisible by these small primes. This approach increases the probability of hitting a valid prime to $1/60$. Upon computing the product, the parties complete the biprimality test as in the Boneh-Franklin test [BF97, BF01].

Upgrading to active security. Our actively secure protocol follows the above blueprint with a few key differences. First, we modify the passive distributed multiplication procedure to consume random instances of Beaver's multiplication triples, given by functionality $\mathcal{F}_{\mathrm{R-Triples}}$ [Bea91]. This modification reduces the overall round complexity and makes it easier to batch zero-knowledge proofs. We obtain active security via an optimized version of the "GMW paradigm" [GMW87]. First, the parties commit to their secret randomness used throughout the protocol. Then, they use efficient zero-knowledge proofs to show that their outgoing messages are consistent with the committed randomness and incoming messages.

We rely on special features of our passive protocol to further optimize the costs of zero-knowledge proofs. In particular, we exploit the fact that RSA modulus generation is a "sampling" functionality that does not involve secret inputs. This allows us to prove correctness of transcripts leading to a surviving candidate. If a proof verification fails, the entire execution is aborted, and the prover is identified as being a cheater. Due to our concrete choices of AHE and proof system, our zero-knowledge proofs are considerably more efficient than those employed in previous related works [HMRT12, FLOP18, CCD+20].

Instantiating the building blocks. We instantiate our AHE scheme with a packed variant of an encryption scheme based on Ring Learning With Errors (RLWE) [LPR13a, LPR13b]. This batched variant performs homomorphic operations on a vector of plaintexts in parallel. We exploit this feature by packing a vector

of CRT shares, which are later reconstructed into a single share. Similarly to [CCD+20], we leverage the fact that the CRT reconstruction algorithm is a linear procedure that each party can run locally. Our main technical contribution is a precise analysis of the parameters needed to achieve the desired level of security. Our RLWE based threshold AHE is significantly more efficient than previous LWE-based threshold AHE from [BD10] and threshold fully homomorphic encryption from [BGG+18].

Our actively secure protocol relies on the Ligero zero-knowledge proof system [AHIV17], which has proof size square-root in the verification circuit size. While other proof systems can offer better asymptotic proof size [GGPR13, Gro16, BCR+19, BBHR19, XZZ+19], they have higher prover computation or memory costs. Here we devise optimized NP statements for ensuring the correctness of the operations related to the underlying AHE (e.g., encryptions, decryptions and randomness sampling), as well as the correctness of the Boneh-Franklin RSA generation protocol. For the biprimality test, we employ a special-purpose Σ-protocol based on [Sho00] for proving correctness of exponentiations in a hidden-order group. We compose these different proof types by checking the overlapping portions their witnesses.

Importantly, excluding the setup phase, the parties only need to use these proofs with respect to the surviving candidates. The prime factor shares of any eliminated candidate and all associated random coins are revealed for anyone to verify correctness. Overall, our zero-knowledge proofs are roughly 40MB per party; in particular, the theorem statements are in fact much larger (see Table 4).

Modular analysis via certified triples. To facilitate a modular description and composable security analysis of our main protocol, we introduce and efficiently realize a "certified triples" functionality $\mathcal{F}_{\mathsf{ctriple}}^{T}$. This functionality naturally extends the "certified OT" functionality from [IKO+11, HIV17] to the arithmetic setting and allows parties to obtain multiplication triples that are guaranteed to satisfy some global relation. (See Figure 4 for a more precise specification.) We present an efficient UC-secure implementation of $\mathcal{F}_{\mathsf{ctriple}}^{T}$ using any threshold AHE with security against semi-malicious adversaries, and then modularly analyze the security of our main protocol in the $\mathcal{F}_{\mathsf{ctriple}}^{T}$-hybrid model.

Prior work We briefly compare against a few recent works in this area. Frederiksen et. al. [FLOP18] present a 2-party protocol for RSA generation using OT-based multipliers. Their protocol has a weaker ideal functionality in that it leaks much more information about the prime factors and becomes inefficient when supporting the standard security with abort notion; this is due to their sieving techniques and the inability to distinguish between cheating and sampling failures. They report on a passive implementation for 2 parties that takes 35 seconds. Chen et al [CCD+20] present an improved approach which does not suffer the security leakage, extends the protocol from 2 parties to n parties, and has a modular security analysis, and can be based only on OT. Our protocol relies on their protocol techniques, but improves by (a) developing a different, more efficient multiplier, (b) introduces the coordinator model, and (c) achieves malicious security using a purely zero-knowledge technique, and (d) reports both a passive and actively secure implementation for up to 4000 parties.

1.3 Related Work

We recount the main prior works about distributed RSA key generation. The seminal work by Boneh and Franklin [BF97, BF01] initiated the line of research on concretely efficient protocols for this task. They introduced the first non-trivial technique for choosing an RSA composite and verifying that it is a biprime. Based on this method, they proposed a protocol in the multi-party honest-majority setting with security against passive (semi-honest) adversaries. Two followup papers [FMY98, NS10], still in the honest-majority setting, strengthened this result and obtained security against *active* (malicious) adversaries. Additional solutions for testing primality in the multi-party setting appear in [ACS02, DM10]. Unlike previous approaches that relied on the biprimality testing procedure from [BF01], these works showed how to securely implement the Miller-Rabin test when the primes are additively shared.

Security in the presence of a dishonest majority in the two-party case poses additional challenges even with only passive corruptions. Cocks [Coc97] initiated the study of the shared generation of the RSA composite in the two-party semi-honest model. Nevertheless, his proposed protocol was later found to be insecure [Cop97, BBBG99]. The problem was solved by Gilboa [Gil99] who presented a protocol in the passive two-party setting, adapting the technique from [BF01]. This work also introduced the technique for secure multiplication based on oblivious transfer, which is now used extensively for secure arithmetic multiplication.

Blackburn et al. [BBBG99] presented the first protocol for the active two-party setting, but did not provide a proof of security. Concurrently, Poupard and Stern [PS98] proposed a provable protocol. However, its running time scaled linearly (rather than logarithmically) with the domain from which the primes are sampled, and moreover there was some leakage about the primes to the adversary. The first fully secure and concretely efficient RSA key generation in the active two-party setting was given by Hazay et al. [HMRT12]. This work further provided the first implementation of such a protocol in the passive two-party setting as well as an extension to the multi-party setting. Finally, based on more recent techniques such as fast, actively secure OT extension, Frederiksen et al. [FLOP18] have improved the state-of-the-art in the two-party setting with active security and also provided an implementation. Their protocol takes on average 35 seconds with 64GB RAM machines and 40Gbps network. Finally, a recent work of Chen et al. [CCD+20] further improves this protocol (without an implementation) and extends it to more than two parties. This protocol can be competitive for a small number of parties when using OT as the underlying primitive, but for medium to large parties such an implementation would scale quadratically and quickly deteriorate.

Beyond the concrete context of RSA modulus generation, there has a been a large body of work on scaling general-purpose secure computation protocols to a large number of parties. Relevant protocols can be divided into ones that involve communication for every gate (or "block" of gates) in a Boolean or arithmetic circuits, and ones in which the communication complexity is independent of the circuit size. Protocols of the latter type, originating from [Gen09, AJL+12], employ fully homomorphic encryption and are not yet competitive for most practical tasks. Protocols that follow the gate-by-gate approach have been optimized for the case of many parties in both the honest-majority and dishonest-majority setting. Assuming a (strict) honest majority, share-packing techniques [FMY98] can be used to make the *total* communication complexity scale only polylogarithmically with the number of parties [DI06, DIK10, BBC+19] for "typical" circuits. Other related works in the honest majority setting include [DN07, BH08, BFO12, BCP15, GLS19]. More relevant to the present work, efficient "gate-by-gate" multi-party protocols for the *dishonest majority* setting were presented in [FH96, CDN01, DPSZ12, KPR18]. The total communication complexity in all of these protocols scales roughly linearly with the number of parties (up to polylogarithmic factors). This should be contrasted with protocols based on oblivious transfer (OT) [GMW87, Kil88, IPS08, KOS16], where the communication grows quadratically with the number of parties. Our work follows the approach initiated by Franklin and Haber [FH96] that relies on *threshold additively-homomorphic encryption* to avoid the quadratic overhead of OT-based protocols.

2 Preliminaries

Basic notations. We let κ be the computational security parameter and s the statistical security parameter. We denote by n the number of parties and by ℓ the bit-length of each prime factor of the RSA modulus. We denote by $[a]$ the set of integers $\{1, \ldots, a\}$. We say that a function $\mu : \mathbb{N} \to \mathbb{N}$ is *negligible* if for every positive polynomial $p(\cdot)$ and all sufficiently large κ's it holds that $\mu(\kappa) < \frac{1}{p(\kappa)}$. We use the abbreviation PPT to denote probabilistic polynomial-time and the bold notation $\boldsymbol{a}$ for denoting vectors.

2.1 Chinese Remainder Theorem (CRT)

First, we recall the beloved (generalized) Chinese Remainder Theorem from Katz and Lindell's textbook *Introduction to Modern Cryptography* [KL14].

Theorem 1 (Chinese Remainder Theorem (CRT))**.** *Let $p_1, p_2, \ldots, p_\ell$ be pairwise relatively prime, ie* $\mathrm{GCD}(p_j, p_j) = 1$ *for all $i \neq j$. Let $N = \Pi_{i=1}^{\ell} p_j$. Then,*

$$\mathbb{Z}_N \simeq \mathbb{Z}_{p_1} \times \ldots \times \mathbb{Z}_{p_\ell} \text{ and } \mathbb{Z}_N^* \simeq \mathbb{Z}_{p_1}^* \times \ldots \times \mathbb{Z}_{p_\ell}^*.$$

Moreover, let f be the function mapping elements $x \in \{0, \ldots, N-1\}$ to tuples $(x_{p_1}, \ldots, x_{p_\ell})$ with $x_{p_j} \in \{0, \ldots, p_j - 1\}$ defined by

$$f(x) = ([x \bmod p_1], \ldots, [x \bmod p_\ell]).$$

Then f is an isomorphism from $\mathbb{Z}_N$ to $\mathbb{Z}_{p_1} \times \ldots \times \mathbb{Z}_{p_\ell}$ as well as an isomorphism from $\mathbb{Z}_N^*$ to $\mathbb{Z}_{p_1}^* \times \ldots \times \mathbb{Z}_{p_\ell}^*$ (where the inputs of f are restricted to $\mathbb{Z}_N^*$).

The linear CRT algorithm which computes f^{-1} is standard and omitted here for space. Next, we give the linear CRT algorithm, which computes f^{-1}. To convert an element represented as $(x_{p_1}, \ldots, x_{p_\ell})$ to its representation mod N:

Algorithm 1 (Linear CRT Algorithm)**.** *Let $(x_{p_1}, \ldots, x_{p_\ell})$, $\{p_1, \ldots, p_\ell\}$, and N be defined as above.*

1. *For all $i \in \{1, \ldots, \ell\}$, compute $N_j = N/p_j$, and find M_j satisfying $N_j \cdot M_j \equiv 1 \bmod p_j$.*
2. *Compute $x := [\sum_{i=1}^{\ell} x_{p_j} N_j M_j \bmod N]$.*

Note that we can compute N_j, M_j in preprocessing.

2.2 Public Key Encryption Schemes (PKE)

We specify first the definitions of public key encryption and IND-CPA.

Definition 1 (PKE)**.** *We say that $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is a* public key encryption scheme *if* $\mathsf{Gen}, \mathsf{Enc}$, *and* Dec *are polynomial-time algorithms specified as follows:*

- Gen*, given a security parameter 1^κ, outputs keys $(\mathrm{PK}, \mathrm{SK})$, where* PK *is a public key and* SK *is a secret key. We denote this by $(\mathrm{PK}, \mathrm{SK}) \leftarrow \mathsf{Gen}(1^\kappa)$.*
- Enc*, given the public key* PK *and a plaintext message m, outputs a ciphertext c encrypting m. We denote this by $c \leftarrow \mathsf{Enc}_{\mathrm{PK}}(m)$; and when emphasizing the randomness r used for encryption, we denote this by $c \leftarrow \mathsf{Enc}_{\mathrm{PK}}(m; r)$.*
- Dec*, given the public key* PK*, secret key* SK *and a ciphertext c, outputs a plaintext message m or an error. We denote this by $m \leftarrow \mathsf{Dec}_{\mathrm{PK},\mathrm{SK}}(c)$.*

For a public key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and a non-uniform adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, we consider the following *IND-CPA game*:

$(\mathrm{PK}, \mathrm{SK}) \leftarrow \mathsf{Gen}(1^\kappa)$.
$(m_0, m_1, history) \leftarrow \mathcal{A}_1(\mathrm{PK})$, s.t. $|m_0| = |m_1|$.
$c \leftarrow \mathsf{Enc}_{\mathrm{PK}}(m_b)$, where $b \leftarrow \{0, 1\}$.
$b' \leftarrow \mathcal{A}_2(c, history)$.
$\mathcal{A}$ wins if $b' = b$.

Denote by $\mathrm{Adv}_{\Pi,\mathcal{A}}(\kappa)$ the probability that $\mathcal{A}$ wins the IND-CPA game.

Definition 2 (IND-CPA)**.** *A public key encryption scheme* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *has* indistinguishable encryptions under chosen plaintext attacks *(IND-CPA), if for every non-uniform adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *there exists a negligible function* negl *such that* $\mathrm{Adv}_{\Pi,\mathcal{A}}(\kappa) \leq \frac{1}{2} + \mathsf{negl}(\kappa)$.

2.3 Threshold Homomorphic Encryption

All definitions are parameterized by a security parameter κ and an integer number of players n. All algorithms take κ and n as input, possibly together with some additional common parameters, which may be a function of κ, n. For notational simplicity, we leave these parameters implicit and provide definitions for the case of n-out-of-n secret sharing. Definitions are easily generalized to arbitrary thresholds t (or arbitrary access structures) by including t as an additional parameter.

Definition 3 (THE)**.** *We say that* $(\mathsf{Gen}, \mathsf{Pub}, \mathsf{Eval}, \mathsf{Dec}, \mathsf{Rec})$ *is a* Threshold Homomorphic Encryption scheme *if* $\mathsf{Pub}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Dec}, \mathsf{Rec}$ *are polynomial time algorithms specified as follows:*

- Gen *is a* randomized *algorithm that on input an integer* $i \in \{1, \ldots, n\}$*, outputs a pair* $(\mathrm{PK}_i, \mathrm{SK}_i) \leftarrow \mathsf{Gen}(i)$ *of public/secret key shares. Without loss of generality, one may assume that the secret key share* SK_i *is the randomness* r_i *used by the key generation algorithm* $(\mathrm{PK}_i, r) = \mathsf{Gen}(i; r)$*, and it is often convenient to think of key generation as consisting of a probabilistic algorithm to sample the secret key share* $\mathrm{SK}_i \leftarrow \mathsf{Sec}(i)$*, together with a* deterministic *algorithm to derive the public key share* $\mathrm{PK}_i = \mathsf{Gen}(\mathrm{SK}_i)$.

- Pub *is a* deterministic *algorithm that on input public key shares* PK_i*, produces*[1] *a public key* $\mathrm{PK} = \mathsf{Pub}(\mathrm{PK}_1, \ldots, \mathrm{PK}_n)$. *More generally, one may consider multi-round key generation algorithms, where, for* $i = 1, \ldots, n$ *and* $r = 1, 2, \ldots$*, one computes a sequence of public shares*

$$\mathrm{PK}_i[r] = \mathsf{Gen}(\mathrm{SK}_i, [\mathrm{PK}[1], \ldots, \mathrm{PK}[r-1]])$$

(starting with $\mathrm{PK}_i[1] = \mathsf{Gen}(\mathrm{SK}_i, [])$*) and round keys*

$$\mathrm{PK}[r] = \mathsf{Pub}(\mathrm{PK}_1[r], \ldots, \mathrm{PK}_n[r]).$$

Then, the public key PK *is set to (a deterministic function of) the concatenation* $(\mathrm{PK}[1], \ldots, \mathrm{PK}[r])$ *of the keys produced in each round.*

- Eval *is a* randomized *algorithm that on input the public key* PK*, an integer* $k \geq 0$*, the description of a function* $f\colon M^k \to M$ *(possibly from a restricted set of possible functions) and a list of* k *ciphertexts* $c_1, \ldots, c_k$*, outputs a new ciphertext* $c \leftarrow \mathsf{Eval}(\mathrm{PK}, k, f, [c_1, \ldots, c_k])$

- *As a special case, encryption of a message* m *is modeled by the evaluation*

$$\mathsf{Enc}(\mathrm{PK}, m) = \mathsf{Eval}(\mathrm{PK}, 0, f(), [])$$

of a 0*-ary function* $f\colon M^0 \to M$*, that on input an empty list of messages, outputs a fixed value* $f() = m$.

- Dec *is a* randomized *algorithm that on input a secret share* SK_i *and ciphertext* c*, outputs a message share* $m_i = \mathsf{Dec}(\mathrm{SK}_i, c)$

[1] As far as security is concerned, the public key may be just the concatenation of the shares $\mathsf{Pub}(\mathrm{PK}_1, \ldots, \mathrm{PK}_n) = (\mathrm{PK}_1, \ldots, \mathrm{PK}_n)$. But it is usually possible to combine these shares into a more compact public key.

- Rec *is a* deterministic *algorithm that on input all message shares* $m_1, \ldots, m_n$*, reconstructs the output message* $m = \mathsf{Rec}(\mathrm{PK}, m_1, \ldots, m_n)$.

We write $\mathsf{Eval}(\mathrm{PK}, k, f, \boldsymbol{c}; r)$ when we want to emphasize the randomness used by the encryption or evaluation algorithm. This randomness r may be chosen uniformly at random, or according to some other efficiently sampleable distribution. Non-uniform distributions are useful in the semimalicious settings, where the adversary may choose the value of the sample r, rather than the randomness used by the sampling algorithm. For simplicity, we leave these sampling algorithms implicit in the definition. In lattice based schemes r is often chosen as a vector with (truncated) discrete gaussian distribution.

Functions f provided to the evaluation algorithm may take any number of arguments $k \geq 0$, but different schemes may support different, restricted sets of functions f. The distribution of the randomness r used by the evaluation algorithm may depend on the function f. The evaluation of some functions f may be deterministic, in which case the randomness r is ignored by Eval.

A Threshold Homomorphic Encryption scheme is usually employed as follows:

1. A client C communicates with n independent servers $S[i]$, a fraction of which may be corrupted in a honest-but-curious or semimalicious manner.

2. Each server $S[i]$ locally generates a secret key SK_i and sends $\mathrm{PK}_i = \mathsf{Gen}(\mathrm{SK}_i)$ to the client C

3. The client C reconstructs the public key $\mathrm{PK} = \mathsf{Pub}(\mathrm{PK}_1, \ldots, \mathrm{PK}_n)$ from the public shares using Pub. In the case of a multiround key generation algorithm, the clients keep computing the values $\mathrm{PK}_i[r] = \mathsf{Gen}(\mathrm{SK}_i, [\mathrm{PK}[1], \ldots, \mathrm{PK}[r-1]])$ and sending them to the server which replies with $\mathrm{PK}[r] = \mathsf{Pub}(\mathrm{PK}_1[r], \ldots, \mathrm{PK}_n[r])$.

4. The client may encrypt message m using $\mathsf{Enc}(\mathrm{PK}, m) = \mathsf{Eval}(\mathrm{PK}, 0, f() = m, [])$

5. The client may homomorphically compute on messages using $\mathsf{Eval}(\mathrm{PK}, k, f, [c_1, \ldots, c_k])$ for any function f supported by the scheme and previously computed (or freshly encrypted) ciphertexts.

6. The client may ask the server to decrypt a previously computed ciphertext c. In response, each server locally computes $\mathsf{Dec}(i, \mathrm{SK}_i, c)$ and sends the output to C. The output of decryption is produced by combining the partial decryptions with Rec.

Correctness of a THE scheme is defined in the obvious way via a game where a client issues a sequence of evaluation queries, including queries with 0-ary functions to encrypt new messages.

Definition 4. *Let* $(\mathrm{PK}_i, \mathrm{SK}_i) \leftarrow \mathsf{Gen}(i)$ *and* $\mathrm{PK} = \mathsf{Pub}(\mathrm{PK}_1, \ldots, \mathrm{PK}_n)$ *be chosen using the key generation algorithms. A THE scheme is correct if for any valid sequence of evaluation queries*

$$c \leftarrow \mathsf{Eval}(\mathrm{PK}, k, f, [c_1, \ldots, c_k])$$

we have

$$\mathsf{Rec}(\mathrm{PK}, \mathsf{Dec}(1, \mathrm{SK}_1, c), \ldots, \mathsf{Dec}(n, \mathrm{SK}_n, c)) = m(c)$$

with probability 1 *(over the randomness used by the key generation, evaluation and decryption algorithms), where the input ciphertexts* c_i *to each query are the result of some previous query in the sequence, and each ciphertext* c *(produced by a query) is associated to a message* $m(c)$ *as follows:*

- *If* $k = 0$*, then* $m(c) = f()$ *is the message being encrypted.*
- *If* $k \geq 1$*, then* $m(c) = f(m(c_1), \ldots, m(c_k))$ *is the result of applying function* f *to the messages associated to the input ciphertexts.*

A scheme may be restricted to specific sequences of encryption/evaluation queries. For example, standard lattice based cryptosystems are only bounded-homomorphic, supporting the evaluation of bounded sequences of operations.

All the above definitions can be extended to functions $f : \prod_i M_i \to M$ with input messages from different sets. But this is not needed in our protocol, so we focus on the simpler definition.

We now define security of a THE scheme. We provide a simple indistinguishability based definition. We focus on security against non-adaptive semi-malicious adversaries, as these are the type needed to obtain full (malicious) security using zero-knowledge proofs. The definition is easily extended to adaptive attacks. We remark that our THE construction and proof of security (see Theorem 5) only uses non-adaptivity during key generation, and this is necessary to achieve security. (If adversary can choose semimalicious secret shares SK_i after seeing the honest public share PK_h, it can easily bias the value of the public key, and easily break the protocol. But once the public key has been chosen, the proof of security of Theorem 5 works also for adversaries that issue their queries adaptively.

Definition 5. *A THE scheme is secure against non-adaptive semi-malicious attacks if any efficient (probabilistic polynomial time) adversary $\mathcal{A}$ has only a negligible advantage in the following game. The game is parameterized by a bit $b \in \{0,1\}$, and consists of the following steps:*

1. *The adversary $\mathcal{A}$ selects the index[2] of an honest party $h \in \{1,\ldots,n\}$, secret key shares $\{\mathrm{SK}_i\}_{i\neq h}$ for all other parties, and a sequence of queries Q described below.[3]*

2. *$\mathrm{SK}_h \leftarrow \mathsf{Sec}(h)$ is chosen at random, and $\mathrm{PK}_h = \mathsf{Gen}(\mathrm{SK}_h)$ is given to $\mathcal{A}$. The adversary can compute the public key $\mathrm{PK} = \mathsf{Pub}(\mathrm{PK}_1,\ldots,\mathrm{PK}_n)$ on its own. For multi-round key generation, the adversary is given honest public key shares $\mathrm{PK}_h[r] = \mathsf{Gen}(\mathrm{SK}_h, [\mathrm{PK}[1],\ldots,\mathrm{PK}[r-1]])$ for every round r, where $\mathrm{PK}[r] = \mathsf{Pub}(\mathrm{PK}_1[r],\ldots,\mathrm{PK}_n[r])$.*

3. *The adversary's queries Q are answered in sequence $q = 1, 2, \ldots$ as follows. There are three types of queries. Challenge and semimalicious queries define a pair of messages $m_0[q], m_1[q]$ and a ciphertext $c[q]$. Challenge and decryption queries produce an output which is returned to the adversary. Semimalicious queries produce no output, and are issued for the sole purpose of defining $m_0[q], m_1[q]$ and $c[q]$.*

   ***Challenge queries** consist of two functions $f_0, f_1 \colon M^k \to M$ and a list of indexes $[i_1,\ldots,i_k]$. These are used to compute*

$$\begin{aligned} m_0[q] &= f_0(m_0[i_1],\ldots,m_0[i_k]) \\ m_1[q] &= f_1(m_1[i_1],\ldots,m_1[i_k]) \\ c[q] &= \mathsf{Eval}(\mathrm{PK}, k, f_b, c[i_1],\ldots,c[i_k]). \end{aligned}$$

   The ciphertext $c[q]$ is returned to the adversary.

   ***Semimalicious queries** consist of a single function $f \colon M^k \to M$, an index list $[i_1,\ldots,i_k]$, and randomness r. These are used to compute*

$$\begin{aligned} m_0[q] &= f(m_0[i_1],\ldots,m_0[i_k]) \\ m_1[q] &= f(m_1[i_1],\ldots,m_1[i_k]) \\ c[q] &= \mathsf{Eval}(\mathrm{PK}, k, f, c[i_1],\ldots,c[i_k]; r). \end{aligned}$$

[2] More generally, for general threshold t, the adversary selects a subset of honest parties.

[3] Specifying all queries at the outset is what makes the definition *non-adaptive*. In a fully adaptive definition the adversary can choose each query after receiving the answer to previous queries. The non-adaptive security definition is enough for our purposes as the protocol is ultimately made secure against active attacks by letting each party commit to its randomness at the outset of the execution, and then behaving deterministically, proving that it followed the protocol in ZK.

Notice that the adversary can compute $c[q]$ on its own, because it knows the randomness r and the list of previous ciphertexts. These queries are useful to generate ciphertexts $c[q]$ which may be referred to in subsequent honest evaluation and decryption queries. Notice also that the two messages $m_0[q], m_1[q]$ may be different even if they are computed using the same function f.

Decryption queries *consist of just an index q, subject to the requirement that $m_0[q] = m_1[q]$. The query is answered with (a sample from)* $\mathsf{Dec}(h, \mathrm{SK}_h, c[q])$.

After receiving the answers to all queries, $\mathcal{A}$ outputs a bit $b' \in \{0,1\}$. The advantage of $\mathcal{A}$ is defined as $|\Pr\{1 \leftarrow \mathcal{A} \mid b = 0\} - \Pr\{1 \leftarrow \mathcal{A} \mid b = 1\}|$.

In decryption queries, after receiving $\mathsf{Dec}(h, \mathrm{SK}_h, c[q])$, the adversary can compute $\mathsf{Dec}(i, \mathrm{SK}_i, c[q])$ for all other $i \neq h$ on its own, and recover the message $m_b[q] = \mathsf{Rec}(\mathrm{PK}, x_1, \ldots, x_n) = m_b[q]$. This provides no information about b because decryption queries are allowed only when $m_0[q] = m_1[q]$. However, the decryption share $\mathsf{Dec}(h, \mathrm{SK}_h, c[q])$ may provide additional information about the secret SK_h.

Similarly to Definition 4, The security definition may be restricted to subsets of valid query sequences, e.g., sequences of bounded length, or sequences where the adversary is allowed a single decryption query at the end of the execution.

2.4 Σ-Protocols

A Σ-protocol is a special zero-knowledge protocol where indistinguishability only holds for honest verifiers. Compiling a Σ-protocol into a zero-knowledge system against an arbitrary verifier can be achieved either using commitment schemes (where the verifier is committing to its challenge before the prover sends its first message and then decommits to as part of the third message), or a random oracle.

Definition 6 (Σ-protocol)**.** *A protocol π is a Σ-protocol between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ for relation $\mathcal{R}$ if it is a 3-round public-coin protocol and the following requirements hold:*

- COMPLETENESS: *If $\mathcal{P}$ and $\mathcal{V}$ follow the protocol on input x and private input ω to $\mathcal{P}$ where $(x, \omega) \in \mathcal{R}$, then $\mathcal{V}$ always accepts.*
- SPECIAL SOUNDNESS: *There exists a polynomial-time algorithm $\mathcal{E}$ that given any x and any pair of accepting transcripts $(a, e, z), (a, e', z')$ on input x, where $e \neq e'$, outputs ω such that $(x, \omega) \in \mathcal{R}$.*
- SPECIAL HONEST-VERIFIER ZERO KNOWLEDGE: *There exists a* PPT *algorithm $\mathcal{M}$ such that*

$$\{\langle \mathcal{P}(x, \omega), \mathcal{V}(x, e)\rangle\}_{x \in \mathcal{L}} \equiv \{\mathcal{M}(x, e)\}_{x \in \mathcal{L}}$$

where $\mathcal{M}(x, e)$ denotes the output of $\mathcal{M}$ upon input x and e, and $\langle \mathcal{P}(x, \omega), \mathcal{V}(x, e)\rangle$ denotes the output transcript of an execution between $\mathcal{P}$ and $\mathcal{V}$, where $\mathcal{P}$ has input (x, ω), $\mathcal{V}$ has input x, and $\mathcal{V}$'s random tape (determining its query) equals e.

2.5 Beaver's Triples

We recall Beaver's multiplication triples functionality $\mathcal{F}_{\text{Triples}}$ that takes additive shares of two secrets over some field $\mathbb{F}$ and produces additive shares of their product. In fact, we will reduce $\mathcal{F}_{\text{Triples}}$ to a random instance $\mathcal{F}_{\text{R-Triples}}$ where all shares are chosen uniformly at random from $\mathbb{F}$. The reduction, due to Beaver [Bea91], is as follows. Denote by $[a]$ the additive sharing of some value $a \in \mathbb{F}$, namely, the tuple $(a_1, \ldots, a_m)$. Then, given a random triple $[a], [b], [c]$ obtained as the output of $\mathcal{F}_{\text{R-Triples}}$, and inputs $[x], [y]$ for $\mathcal{F}_{\text{Triples}}$, we can compute $[xy]$ by first reconstructing $e = [x - a]$ and $d = [y - b]$. Next, the parties compute a (trivial) secret sharing $[ed]$ of ed by having P_1 set its share to ed, and the rest of the parties set their

shares to 0. Finally, the parties compute the following equation (each party locally computes the equation on its own shares)

$$\begin{aligned}[xy] &= [c] + e[y] + d[x] - [ed] \\ &= [ab] + (x-a)[y] + (y-b)[x] - (x-a)(y-b).\end{aligned}$$

2.6 The RSA Composite Functionality

Functionality $\mathcal{F}_{\text{RSA}-\text{ML}}$, Figure 1, captures the distributed generation of n parties of the RSA composite in the active (malicious) setting by allowing corrupted parties to arbitrarily choose their shares and restart the execution. This models a corrupted party that aborts after seeing the composite. In case of an abort, the functionality reveals the factorization of the discarded composite and restarts. Looking ahead, we require such a modeling in order to conclude the simulation successfully, as our active protocol only catches deviations at the last round of the protocol; see more discussion in Section 4.

Functionality $\mathcal{F}_{\text{RSA}-\text{ML}}$

Functionality $\mathcal{F}_{\text{RSA}-\text{ML}}$ communicates with parties $P_1, \ldots, P_n$, a coordinator C and an adversary $\mathcal{S}$ corrupting a subset $U \subset [n]$. $\mathcal{F}_{\text{RSA}-\text{ML}}$ is parameterized by parameter ℓ.

1. **Adversary shares.** Upon receiving a message $(\mathsf{shares}, sid, ssid, P_j, p_j, q_j)$ for $j \in U$ from the adversary, record the tuple $(sid, ssid, P_j, p_j, q_j)$ and send message $(\mathsf{receipt}, sid, ssid, P_j)$ to P_j and $\mathcal{S}$.

2. **Shares generation.** Upon receiving a messages $(\mathsf{sample}, sid, ssid, P_j)$ from the honest P_j, record the tuple $(sid, ssid, P_j)$ send message $(\mathsf{receipt}, sid, ssid, P_j)$ to P_j and $\mathcal{S}$.

   Once tuple $(sid, ssid, P_j)$ has been recorded from all parties P_j, sample random ℓ-bit values for the honest parties' shares $\{p_j, q_j\}_{j \notin U}$ satisfying

   - $p_1 \equiv q_1 \equiv 3 \bmod 4$
   - $p_2, \ldots, p_n \equiv q_2 \ldots, q_n \equiv 0 \bmod 4$
   - $p = \sum_{j=1}^{n} p_j$ and $q = \sum_{j=1}^{n} q_j$ are primes

3. **Output.** Send $(\mathsf{output}, N = pq)$ to all P_j and $\mathcal{S}$.

4. **Identify Cheater.** If the functionality receives $(\mathsf{corrupt}, sid, ssid, P_j)$, it records $(\mathsf{cheater}, sid, P_j)$ and sends $(\mathsf{factors}, sid, ssid, p, q)$ to $\mathcal{S}$.

5. **Certify.** Upon receiving $(\mathsf{certify}, sid, ssid, P_j)$ from all P_j, check if $(\mathsf{cheater}, sid, P_j)$ exists in memory. If so, the functionality aborts the execution and restarts with the same set of parties excluding P_j.

Figure 1: The shared RSA functionality in the active setting.

2.7 Commit-and-Prove Ideal Functionality

[CLOS02] gives the following one-to-many commit-and-prove ideal functionality, Figure 2, which will be used for certifying our triples generation.

Functionality $\mathcal{F}_{\text{CP}}$

Functionality $\mathcal{F}_{\text{CP}}$ communicates with parties $P_1, \ldots, P_n$, a coordinator C and an adversary $\mathcal{S}$ who corrupts the subset of parties in $U \subset [n]$. $\mathcal{F}_{\text{CP}}$ is parameterized by an NP relation $\mathcal{R}$ and an integer t.

- **Commit phase:** Upon receiving a message $(\mathsf{commit}, sid, ssid, P_j, w)$ from P_j where $w \in \{0,1\}^t$, record the value w within $\bar{w}$ and send the message $(\mathsf{recv\text{-}commit}, sid, ssid, P_j)$ to C and $\mathcal{S}$. (Initially, the list $\bar{w}$ is empty.)

- **Prove phase:** Upon receiving a message $(\mathsf{CP\text{-}prover}, sid, ssid, P_j, x)$ from P_j where $x \in \{0,1\}^{\mathsf{poly}(t)}$, compute $\mathcal{R}(x, \bar{w})$: If $\mathcal{R}(x, \bar{w}) = 1$, then send the message $(\mathsf{CP\text{-}proof}, sid, ssid, x)$ to C and $\mathcal{S}$. Otherwise, ignore.

Figure 2: Functionality for commit-and-prove.

3 Certified Triples Functionality

A core building block in our protocol is a functionality that generates multiplication triples (or Beaver triples [Bea91]) (see Section 2.5). In this section, we introduce an extension of this functionality which generates triples and allows the parties to prove at a later point a global relation over their individual triples (Figure 4).

The certified triples functionality helps abstract the public key cryptographic primitive that we rely on, namely a threshold additively homomorphic encryption with security against semi-malicious adversaries. Furthermore, it allows us to modularly analyze the security of the main protocol assuming ideal access to the $\mathcal{F}_{\mathsf{ctriple}}$ functionality. In more detail, this functionality allows for the parties to first obtain multiplication triples and later certify their actions w.r.t some global relation on the triples. This is similar to the certified oblivious transfer functionality that allows the sender in a oblivious transfer (OT) protocol [IKO+11] to certify its inputs to the OT w.r.t. some global NP relation. Our abstraction also supports identifiable abort, where the functionality identifies the party that failed the execution. In our protocol, if a party deviates, its identity is revealed to all parties. The complete protocol is shown in Figure 4.

Next, we realize our functionality via a threshold (additively) homomorphic encryption (THE) scheme. We give the security definition for THE in Section 2.3. We discuss implementing these functionalities using Ring-LWE in Section 6.1.

Functionality $\mathcal{F}_{\mathsf{ctriple}}^{T}$

Functionality $\mathcal{F}_{\mathsf{ctriple}}$ communicates with parties $P_1, \ldots, P_n$, coordinator C and an adversary $\mathcal{S}$ who corrupts the subset of parties in $U \subset [n]$. $\mathcal{F}_{\mathsf{ctriple}}$ is parameterized by an NP relation $\mathcal{R}_{\text{EXT}}$, an integer T, and domains $B_1, \ldots, B_T$. The functionality generates T triples where the i^{th} triple is over the finite field $\mathbb{F}_{B_i}$.

Triples generation phase: Upon receiving a message $(\mathsf{generate}, sid, ssid, P_j)$, record the tuple $(ssid, P_j)$ and send the message $(\mathsf{receipt}, sid, ssid, P_j)$ to P_j and S. Upon receiving a message $(\mathsf{generate}, sid, ssid, \mathcal{S}, \{\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j\}_{j\in U})$ from S record $(ssid, \{\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j\}_{j\in U})$. Once a tuple $(ssid, P_j)$ has been received from all parties, sample $\{\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j\}_{j\notin U}$ at random from $\mathbb{F}_{B_i}$ conditioned on the following equation

$$\sum_{j=1}^{n} c_j^i = \left(\sum_{j=1}^{n} a_j^i\right) \cdot \left(\sum_{j=1}^{n} b_j^i\right)$$

for all $i \in [T]$ where $\boldsymbol{x}_j = (x_j^1, \ldots, x_j^T)$ for $\boldsymbol{x} \in \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\}$.

Abort: If the functionality receives $(\mathsf{abort}, sid, ssid, P_j)$ for $j \in U$ before any $\mathsf{generate}$ message was received, it ignores all messages in the **Triples generation phase**. If after receiving an abort message it receives $(\mathsf{assign}, sid, ssid, \mathcal{S}, \{\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j\}_{j\notin U})$ from the adversary, it records $(sid, ssid, \{\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j\}_{j\notin U})$.

Output phase: If a triple is recorded for every party, send $(\mathsf{triple}, sid, ssid, \boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j)$ to party P_j for $j \notin U$.

Certification phase: Upon receiving a message $(\mathsf{certify}, sid, ssid, P_j, (x_j, \omega_j))$, record the tuple $(ssid, P_j, (x_j, \omega_j))$. If no abort message was recorded for P_j, send the message $(\mathsf{verify}, sid, ssid, P_j, \mathcal{R}_{\text{EXT}}(x_j, \omega_j, \boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j))$ to all parties and S. If an abort message was received on behalf of P_j, send $(\mathsf{abort}, sid, ssid, P_j)$ to C.

Figure 3: The certified triples functionality.

PROTOCOL $\Pi_{\mathsf{ctriple}}^{T}$

Notations. The protocol communicates with parties $P_1, \ldots, P_n$ and a coordinator C, is parameterized by modulus M, and relies on a threshold additively-homomorphic encryption scheme $(\mathsf{Gen}, \mathsf{Pub}, \mathsf{Eval}, \mathsf{Dec}, \mathsf{Rec})$ with plaintext ring $\mathbb{R}$, packing factor T, and an r_{Enc}-round key-generation protocol. Let $B_1, \ldots, B_T$ be the corresponding domains from which each of the T vectors is sampled. Let $\mathsf{Com}(\cdot, \cdot)$ be a non-interactive commitment scheme. Let λ denote the security parameter.

Commit phase: Each party P_j commits to its randomness R_j for the protocol by sending $(\mathsf{commit}, sid, ssid, R_j)$ to $\mathcal{F}_{\text{CP}}$. $\mathcal{F}_{\text{CP}}$ sends $(\mathsf{receipt}, sid, ssid)$ to C.

THE setup phase.

1. Party P_j runs $\mathsf{Gen}(j; r_j)$ to obtain $\mathrm{PK}_j[1], \mathrm{SK}_j$ with security parameter λ and number of parties n, where r_j is obtained from the random tape R_j. P_j sends $\mathrm{PK}_j[1]$ to the coordinator. The coordinator C computes $\mathrm{PK}[1] = \mathsf{Pub}(\mathrm{PK}_1[1], \ldots, \mathrm{PK}_n[1])$ and broadcasts to all parties.

2. For $r = 2, \ldots, r_{\mathsf{Enc}}$, party P_j computes $\mathrm{PK}_j[r] = \mathsf{Gen}(\mathrm{SK}_j, \mathrm{PK}[1], \ldots, \mathrm{PK}[r-1])$ and sends it to the coordinator. The coordinator computes $\mathrm{PK}[r] = \mathsf{Pub}(\mathrm{PK}_1[r], \ldots, \mathrm{PK}_n[r])$ and broadcasts to all parties.

3. At the end of r_{Enc}-round, the parties set $\mathrm{PK} = \mathrm{PK}[r_{\mathsf{Enc}}]$.

Input generation phase. Party P_j samples vectors $\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j$, where the ith element of each vector is in $\mathbb{Z}_{B_i}$. Also, P_j samples offsets $z_j[1], \ldots, z_j[T]$ where $z_j[i]$ is sampled uniformly from $[-nB_i 2^\lambda, nB_i 2^\lambda]$. Let $\boldsymbol{z}_j = (z_j[1]B_1, \ldots, z_j[T]B_T)$.

Triples generation phase.

1. Party P_j sends $\alpha_j = \mathsf{Enc}_{\mathrm{PK}}(\boldsymbol{a}_j)$ to C.

2. Upon receiving input from all parties, C computes and broadcasts $\alpha = \mathsf{Eval}(\mathrm{PK}, n, \mathbf{ADD}(\cdot), [\alpha_1, \ldots, \alpha_n])$, where $\mathbf{ADD}(\cdot)$ is the pointwise addition operation on length T vectors.

3. Each party P_j computes $\beta_j = \mathsf{Eval}(\mathrm{PK}, 1, f_{lin}(\boldsymbol{b}_j, \boldsymbol{c}'_j, \cdot), \alpha)$ and sends β_j to C where $\boldsymbol{c}'_j = \boldsymbol{z}_j - \boldsymbol{c}_j$. The function f_{lin} is defined over three inputs: $\boldsymbol{b}_j, \boldsymbol{c}'_j$, and $\boldsymbol{m}$ (a plaintext vector corresponding to α's decrypted ciphertexts). It takes the i^{th} component of $\boldsymbol{b}_j, \boldsymbol{c}_j$ and $\boldsymbol{m}$ (denoted b, c, m) and returns $bm + c$ as the i^{th} component of the output vector.

4. C computes and broadcasts $\beta = \mathsf{Eval}(\mathrm{PK}, n, \mathbf{ADD}(\cdot), [\beta_1, \ldots, \beta_n])$.

5. Each party P_j computes $d_j = \mathsf{Dec}(j, \mathrm{SK}_j, \beta)$ and transmits it to C.

6. C computes $\boldsymbol{w} = \mathsf{Rec}(\mathrm{PK}, d_1, \ldots, d_n)$ and sends $\boldsymbol{w}$ to P_1.

7. P_1 outputs its triples as $(\boldsymbol{a}_1, \boldsymbol{b}_1, \boldsymbol{c}''_1)$ where the i^{th} element of $\boldsymbol{c}''_1$ is set to $c_1^i + w_i \bmod \mathbb{Z}_{B_i}$. All other parties output $(\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j)$ as their triples.

Inputs for certification. Party P_j receives a statement x_j and a witness ω_j from the environment $\mathcal{Z}$. (Recall that this part of the NP statement is used to certify how the triples are used in the larger protocol.)

Generating proofs. Let τ_j be P_j's transcript of interaction with the coordinator. Party P_j sends $(\mathsf{commit}, sid, \omega_j)$ and $(\mathsf{CP\text{-}prover}, sid, (x_j, \tau_j))$ to $\mathcal{F}_{\mathrm{CP}}$. (We instantiate $\mathcal{F}_{\mathrm{CP}}$ with the NP-relation $\mathcal{R}_{\mathrm{CP}}$ that takes as input the statement (x_j, τ_j) and witness $(\omega_j, R_j, \boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j)$. $\mathcal{F}_{\mathrm{CP}}$ outputs 1 if

$$(x_j, (\omega_j, \boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j)) \in \mathcal{R}_{\mathrm{EXT}}$$

and $(\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j)$ are consistent with R_j, and τ_j is consistent with the honest party's code using randomness R_j).

Figure 4: Protocol realizing the certified triples functionality $\mathcal{F}^T_{\mathsf{ctriple}}$.

Theorem 2. *Protocol Π_{ctriple} UC-realizes $\mathcal{F}_{\mathsf{ctriple}}$ in the $(\mathcal{F}_{\mathrm{COM}}, \mathcal{F}_{\mathrm{CP}})$-hybrid model, in the presence of active adversaries.*

Proof. (sketch) Let $\mathcal{A}$ be an active adversary; we construct a simulator $\mathcal{S}$ for the ideal process $\mathcal{F}_{\mathsf{ctriple}}$. Simulator $\mathcal{S}$ internally invokes $\mathcal{A}$ and proceeds as follows:

– Simulating the communication with $\mathcal{Z}$: The input values received by $\mathcal{S}$ from $\mathcal{Z}$ are written on $\mathcal{A}$'s input tape, and the output values of $\mathcal{A}$ are copied to $\mathcal{S}$'s own output tape.

– Simulating the commit phase: $\mathcal{S}$ extracts all the randomness used by the corrupted parties by intercepting the message sent to the $\mathcal{F}_{\text{CP}}$.

– Simulating the **THE setup** phase:

The simulator $\mathcal{S}$ simulates the messages from uncorrupted parties honestly. At the end of the phase, if the simulator observes that any corrupted party P_j sends a message inconsistent with the randomness R_j, it samples random inputs for the honest parties $\{\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j\}_{j \notin U}$ and completes the internal emulation till the end of the **Triples generation phase**. Party P_1's $\boldsymbol{c}_1$ is altered in Step 7 as per the protocol. Then, $\mathcal{S}$ sends (abort, P_j) and $(\mathsf{assign}, sid, ssid, \mathcal{S}, \{\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j\}_{j \notin U})$ to $\mathcal{F}_{\mathsf{ctriple}}$.

– Simulating the **triples generation** phase:

If an abort message has not yet been sent, then the simulator proceeds honestly with uncorrupted parties sampling inputs $\{\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j\}_{j \notin U}$. If all corrupted parties proceed consistently w.r.t. their randomness, the simulator identifies the inputs of the corrupted parties from the randomness extracted in the commit phase and sends $(\mathsf{generate}, sid, ssid, \mathcal{S}, \{\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j\}_{j \in U})$ to $\mathcal{F}_{\mathsf{ctriple}}$.

At the end of the phase, if the simulator observes that any corrupted party P_j sends a message inconsistent with the randomness R_j, then, as in the previous step, it completes the emulation until the end of the **Triples generation phase** and sends (abort, P_j) and $(\mathsf{assign}, sid, ssid, \mathcal{S}, \{\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j\}_{j \notin U})$ to $\mathcal{F}_{\mathsf{ctriple}}$.

– Simulating the **proof generation** phase:

In this phase, the simulator extracts the witness ω_j from the commit command. Then, for every corrupted P_j, it verifies that $\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j$ is consistent with R_j, and R_j is consistent with τ_j. If they are, it sends $(\mathsf{certify}, sid, P_j, (x_j, \omega_j))$ to $\mathcal{F}_{\mathsf{ctriple}}$. Otherwise, it sends (abort, P_j).

Next, we argue correctness of the simulation. We show that for every environment $\mathcal{Z}$ it holds that:

$$\text{IDEAL}_{\mathcal{F}_{\mathsf{ctriple}}, \mathcal{S}, \mathcal{Z}}(n) \approx \text{REAL}^{\mathcal{F}_{\text{COM}}, \mathcal{F}_{\text{CP}}}_{\Pi_{\mathsf{ctriple}}, \mathcal{A}, \mathcal{Z}}(n)$$

In arguing security, there are two main cases:

Case 1: Some corrupted party deviates during the protocol. Namely, its actions are inconsistent with randomness that it committed during the **Commit phase**. In this case, we observe that the simulated transcript is identically distributed to the real world. This is because the simulator sets the inputs of the honest parties as per the real world execution.

Case 2: No corrupted party deviates until the end of the **Triples generation phase**. For this case, we argue that the simulation is computationally indistinguishable to the real world by a reduction to the security of threshold THE scheme. We detail the argument in Appendix D.

■

4 The Actively Secure Protocol

The classic GMW paradigm [GMW87] provides a generic compiler from passive to active security via zero-knowledge proofs and commitments, where zero-knowledge proofs are employed in each round of the protocol to enforce honest behavior in each round. We follow this paradigm with one important difference. After parties commit to their randomness at the beginning of the computation, parties provide only one zero-knowledge proof at the end of the protocol, instead of in each round. We will argue that this suffices to provide full security in our protocol. We note that our approach generalizes to any input-less functionality.

On a high-level, our protocol has four phases (1) Sample primes or pre-sieving, (2) Compute bi-prime, (3) Sieve easy non-bi-primes or post-sieve and (4) Sieve all non-bi-primes. Most of the tasks in the phases can be further reduced to multiple invocations of securely multiplication. Towards this, we use the certified Beaver triples ideal functionality $\mathcal{F}_{\mathsf{ctriple}}$ (Figure 4) that provides the following guarantees: (I) first, sufficiently many Beaver multiplication triples are generated in parallel (Section 2.5 for more details) (II) at a later point, the parties prove an arbitrary predicate over the sampled triples. We analyze our protocol modularly in the $\mathcal{F}_{\mathsf{ctriple}}$-hybrid model.

PROTOCOL $\Pi_{\mathrm{RSA-ML}}$

Notations. Let s be the statistical security parameter, $P_1, \ldots, P_n$ be the set of parties and C is a coordinator. Let T_1, T_2 be natural numbers such that the product $\prod_{j=1}^{T_1} d_j$ is $\ell - 2$ bits and the product $\prod_{j=1}^{T_2} d_j$ is greater than $2\ell - 2$ bits where $d_1, \ldots, d_{T_1}, \ldots, d_{T_2}$ denote the first T_2 primes excluding 2. We bucket the primes $d_1, \ldots, d_{T_1}$ into T buckets of at most m bits and denote the buckets by $\tau_1 \ldots \tau_T$ the products of the primes in the corresponding buckets, i.e. $\prod_{i=1}^{T} \tau_i = \prod_{j=1}^{T_1} d_j$. Finally, let $\mathcal{N}$ be the number of share instances that are sampled and $\mathcal{N}'$ be the number of candidates.

Triples generation. Every Party P_j sends $(\mathsf{generate}, sid, ssid, P_j)$ to $\mathcal{F}_{\mathsf{ctriple}}$ and receives the receipt message for $\mathcal{N}T + \lceil \ell/m \rceil \mathcal{N}' + \lceil (5\ell + 2\log n + s)/m \rceil J_{\mathtt{surv}}$ multiplication triples $[a], [b], [c]$, where $J_{\mathtt{surv}}$ is the number of candidates surviving the Jacobi test and assuming that $\ell = \log_2 \left(\prod_{j=T_1+1}^{T_2} d_j \right) - \log_2 \left(\prod_{j=1}^{T_1} d_j \right)$.

Pre-sieving. The parties consume the first $\mathcal{N}T$ multiplication triples. In detail, for every $i \in \mathcal{N}$ and $t \in [T]$,

- P_j samples $r_{i,t}^j \leftarrow [0, \tau_t - 1]$ and $\tilde{r}_{i,t}^j \leftarrow [0, \tau_t - 1]$.
- Now, parties consume a multiplication triple. Observe that $r_{i,t}^j, \tilde{r}_{i,t}^j$ are additive shares of $x = \sum_j r_{i,t}^j$ and $y = \sum_j \tilde{r}_{i,t}^j$. So each party locally computes $[e] = [x - a] \bmod \tau_t$ and $[d] = [y - b] \bmod \tau_t$ and sends these values to the coordinator C. The coordinator computes $e = \sum_j [e] \bmod \tau_t$ and $d = \sum_j [d] \bmod \tau_t$ and sends e, d to all parties. Next, the parties locally multiply ed and do a (trivial) secret sharing $[ed]$ via P_1 getting ed and all other parties getting 0. Finally, each party locally computes and sends to the coordinator its share

$$\begin{aligned}[xy] &= [c] + e[y] + d[x] - [ed] \bmod \tau_t \\ &= [ab] + (x - a)[y] + (y - b)[x] \\ &\quad - (x - a)(y - b) \bmod \tau_t.\end{aligned}$$

  Upon receiving from all parties' $[xy]$, the coordinator computes $\mathsf{mult}_{i,t} = xy \bmod \tau_t$ and sends back the value to all parties.
- The parties record $\mathsf{mult}_{i,t}$ and conclude with the GCD check: If $\mathrm{GCD}(\mathsf{mult}_{i,t}, \tau_t) = 1$, P_j adds the pair $(r_{i,t}^j, \tilde{r}_{i,t}^j)$ to a list L_t^j.

For all lists L_t, the parties re-index the elements (i.e., the first element has $i = 1$, etc). Furthermore, all lists are trimmed to match the size of the smallest list, namely $\mathcal{N}' = \min_t |L_t|$.

CRT reconstruction. Each party P_j locally computes its shares of the prime candidates by invoking the CRT reconstruction algorithm. For the i^{th} share of the primes, P_j first collects the i^{th} pairs from each of the lists $(r^j_{i,1}, \tilde{r}^j_{i,1}), \ldots, (r^j_{i,T}, \tilde{r}^j_{i,T})$ and sets $p_{i,j}$ and $q_{i,j}$ by respectively applying the CRT construction on the tuples $(r^j_0, r^j_{i,1}, \ldots, r_{j,T})$ and $(\tilde{r}^j_0, \tilde{r}^j_{i,1}, \ldots, \tilde{r}_{j,T})$ w.r.t the modulus $(4, \tau_1, \ldots, \tau_T)$ where P_1 sets $r^0_0 = \tilde{r}^0_0 = 3$ and the rest of the parties set $r^0_j = \tilde{r}^0_j = 0$.

Candidate generation. Next, for $i \in [0, \mathcal{N}']$ the parties compute the candidate RSA modulus $N_i = (\sum_{j=1}^n p_{i,j})(\sum_{j=1}^n q_{i,j})$. We rely on CRT to perform this multiplication. More precisely, the parties bucket primes such that the product in each bucket is at most m-bits as before, but up to T_2 primes. The parties will then deconstruct $p_{i,j}$ and $q_{i,j}$ w.r.t the products in each bucket. Finally, using the multiplication triples consumption technique from **Pre-sieving**, compute the products w.r.t to corresponding modulus and then apply CRT reconstruction.

Jacobi test. The parties execute the following steps s times:

- Using a coin-tossing protocol, the parties sample a random $\gamma_i \in \mathbb{Z}^*_{N_i}$ for each $i \in \mathcal{N}'$. This involves each party sampling a random value modulo N_i and sending it to the coordinator. The coordinator sums these values to obtain γ_i and checks that γ_i has Jacobi symbol 1 over N_i. If so, send γ_i to all parties. Otherwise, they restart and repeat the sampling until they find γ_i that has Jacobi symbol 1 w.r.t N_i.

- Each party sends $\gamma_i^{(-p_{i,j}-q_{i,j})/4} \bmod N_i$ to C. Then, C computes $\gamma_i^{(N_i+1-p_i-q_i)/4} \bmod N_i$ and eliminates candidate N_i if the value is not $\{1, -1\}$.

GCD test. For candidates N_i that pass the Jacobi test, let $V = 2^{3\ell+\log n+s}$ and choose a number $Q_{\mathrm{GCD}} > V \cdot N_i \cdot n$ such that it is a product of m-bit numbers, say $(B_1, \ldots, B_{\lceil \log Q_{\mathrm{GCD}}/m \rceil})$. Parties sample random numbers $a_j \in \mathbb{Z}_{N_i}$ and $v_j \in [-V, V]$. Then P_j maps a_j and $p_{i,j} + q_{i,j}$ into the CRT domain using modular reduction. In each bucket B_k, the parties consume a multiplication triple (as described in **Pre-sieving**) to receive the share $[z_k]$ of $z_k = a \cdot (p_i + q_i - 1) \bmod B_k$. Then, each party computes $[\alpha_k]_j = [z_k] + v_j \cdot N_i \bmod B_k$. Once this is done for all buckets, each P_j locally applies CRT reconstruction to their $[\alpha_k]_j$ and get α_j. Parties send α_j to C, who computes $\alpha = \sum_j \alpha_j \bmod Q_{\mathrm{GCD}}$ and sends α to the parties. Parties locally compute $z = \alpha \bmod N_i$. The parties eliminate N_i if $\mathrm{GCD}(N_i, z) \neq 1$.

Certification and Σ-protocol. If no i survives, parties restart the protocol at **Triples generation**. Otherwise, for the minimum i that survives the biprimality test, every party P_j invokes the **Certification phase** of $\mathcal{F}_{\mathsf{ctriple}}$ with its input $(\mathsf{certify}, sid, ssid, P_j, (x_j, \omega_j))$ to certify that they behaved honestly. The parties proceed based on the response from $\mathcal{F}_{\mathsf{ctriple}}$. Condition on not aborting, the parties further run the Σ-protocol Π_{DL} for proving the knowledge of a discrete log in groups with unknown order [Sho00] (Figure 6).

Output phase. If the certification and Σ-protocol pass for all parties, output $(N_i, p_{i,j}, q_{i,j})$ to P_j. Otherwise, the protocol outputs P_j for parties who fail the above test.

Figure 5: Actively secure generation of an RSA composite.

Security proof. We prove security by describing a simulator and arguing indistinguishability of simulation in the UC-model. Recall that our protocol on a high-level follows the classic GMW paradigm with one modification. Namely, we employ zero-knowledge proofs to certify the actions of a party only once and

at the end of the protocol. The main subtlety that arises in the simulation is simulating messages from the honest party in an indistinguishable way up until the certification if the adversary deviates ahead of the certification. Recall that, the goal of the simulator is to receive a bi-prime from the $\mathcal{F}_{\mathrm{RSA-ML}}$-functionality and embed it within the simulation. In our design, the protocol up until the certification remains secure as long as the adversary remains honest. However, if a bi-prime has been embedded and the adversary deviates, this no longer holds and the simulator still needs to be able to continue simulating honest party messages until the certification (where the protocol will abort as the adversary cannot provide a valid witness for certification).

Our strategy to tackle this is to ensure that the simulator can identify exactly when the adversary deviates. This can be achieved by having the adversary commit to its randomness ahead of the protocol and attest later in the certification that is follows the honest code with the committed randomness. Next, we carefully simulate the honest party messages in such a way that when the adversary deviates, either (1) the simulator will refrain from embedding the target bi-prime (received from $\mathcal{F}_{\mathrm{RSA-ML}}$) and produce a view identically distributed to the real world, or (2) the simulator embeds the bi-prime and when a deviation occurs, the simulator can complete the simulation with the knowledge of the factors of the target bi-prime. We remark that in the second case, the target bi-prime is not secure. We therefore extend the $\mathcal{F}_{\mathrm{RSA-ML}}$ functionality to additionally accommodate a special request from the adversary upon which it will provide the factors of the bi-prime.

In slight more detail, the two cases will depend on exactly where the first deviation by some corrupted party occurs. If it occurs before the end of the triples generation phase, then we will be in case (1) and otherwise case (2).

Theorem 3. *Protocol $\Pi_{\mathrm{RSA-ML}}$ UC-realizes $\mathcal{F}_{\mathrm{RSA-ML}}$ in the $\{\mathcal{F}_{\mathsf{ctriple}}\}$-hybrid model , in the presence of up to $n-1$ active adversaries.*

Proof. Let $\mathcal{A}$ be an active adversary interacting with the parties running the $\Pi_{\mathrm{RSA-ML}}$. We construct an adversary $\mathcal{S}$ for the ideal process for $\mathcal{F}_{\mathrm{RSA-ML}}$ such that no environment $\mathcal{Z}$ can distinguish beyond a negligible probability if it is interacting with $\mathcal{A}$ and the above protocol or with $\mathcal{A}$ in the ideal process for $\mathcal{F}_{\mathrm{RSA-ML}}$. Let U be a coalition of at most $n-1$ parties that $\mathcal{A}$ corrupts. Without loss of generality we assume the coordinator is corrupted passively. $\mathcal{S}$ corrupts the parties in U and proceeds as follows:

1. $\mathcal{S}$ receives an RSA biprime N from $\mathcal{F}_{\mathrm{RSA-ML}}$ and emulates the role of $\mathcal{F}_{\mathsf{ctriple}}$.

2. **Triples generation.** The simulator emulates the role of functionality $\mathcal{F}_{\mathsf{ctriple}}$ and either receives a $(\mathsf{generate}, sid, ssid, \mathcal{A}, \{a_j, b_j, c_j)\}_{j\in U})$ or $(\mathsf{abort}, sid, ssid, P_j)$ from $\mathcal{A}$. In the former case, it records $\{a_j, b_j, c_j)\}_{j\in U}$. In the latter case, it waits until it receives $(\mathsf{assign}, sid, ssid, \mathcal{A}, \{a_j, b_j, c_j)\}_{j\notin U})$ from $\mathcal{A}$ which it records. Furthermore, in the latter case, the simulator uses the triples provided by the adversary to simulate the honest parties for the rest of the protocol (until certification).

3. **Pre-sieving.** If an abort message was not received in the previous phase, it identifies the index i^* to implant the target RSA candidate. In order to maintain the right distribution for the index, the simulator simulates $\mathcal{F}_{\mathsf{ctriple}}$ honestly and samples triples for honest parties. Based on the triples of the honest parties and the corrupted parties (recorded in the previous step). Next, the simulator identifies the first index $i^* \in \mathcal{N}'$ in the candidate generation step that would lead to valid RSA biprime. If no index leads to a biprime, then the simulator proceeds honestly with the sampled triples. If an index i^* is found, the simulator also identifies their indices before the lists L_t^j were re-indexed. These are the positions that need to be manipulated in the pre-sieving stage. For all indices aside from i^*, the messages will be simulated according to the honest algorithm. We continue the rest of the simulation specifically for index i^* as follows:

(a) At index i^*, the simulator manipulates the honest parties' messages as follows.

- To simulate the honest parties' $[e]$ and $[d]$, $\mathcal{S}$ samples them uniformly at random from $[0, \tau_t - 1]$. Afterwards, $\mathcal{S}$ receives the corrupted parties' $[e], [d]$ messages. Observe that after this point, $\mathcal{A}$'s future messages are determined, so $\mathcal{S}$ can detect when $\mathcal{A}$ sends incorrect messages.
- For simulating the honest parties' $[xy]$ messages, $\mathcal{S}$ does the following:
  - $\mathcal{S}$ computes $e = \sum_{[n]}[e]$ and $d = \sum_{[n]}[d]$.
  - Recall that $\mathcal{S}$ knows the corrupted parties' triples $\{[a], [b], [c]\}_{j \in U}$. Then using the $[e], [d]$ from above, compute:

$$\sum_U [x] = e - \sum_{[n]-U} [e] + \sum_{j \in U} [a] \bmod \tau_t$$

    and

$$\sum_U [y] = d - \sum_{[n]-U} [d] + \sum_{j \in U} [b] \bmod \tau_t.$$

  - Now, compute the corrupted

$$\sum_U [xy] = \sum_{j \in U} [c] + e \cdot \sum_U [y] + d \cdot \sum_U [x] - \sum_U [ed] \bmod \tau_t.$$

  - Since the target value is $xy = N \mod \tau_t$, the simulator can simply compute the honest shares $[xy] \bmod \tau_t$ as additive shares of $xy - \sum_U [xy] \bmod \tau_t$.
- Index i^* was chosen so that the GCD test always passes, ie outputs 1.

After every simulated message, $\mathcal{S}$ receives the corrupted parties' corresponding messages and checks that it is consistent with the previous transcript and committed triples. If $\mathcal{A}$ sends the messages honestly, the simulator simply continues to the next message in the simulation. If not, $\mathcal{S}$ follows Step 3b. This means that if the adversary sends all messages in this phase honestly, $\mathcal{S}$ ignores Step 3b and continues to the next phase.

(b) If $\mathcal{A}$ deviates, i.e. sends $[xy]$ messages inconsistent with the committed triples and transcript, for any corrupted party P_j, the simulator (acting as $\mathcal{F}_{\mathsf{ctriple}}$) records $(\mathsf{abort}, sid, ssid, P_j)$ and sends $(\mathsf{corrupt}, sid, ssid, P_j)$ to the functionality $\mathcal{F}_{\mathrm{RSA-ML}}$, receiving the prime factors p, q of N. Then the adversary completes the rest of the simulation for index i^* using p and q. Specifically, we update the honest parties' prime shares $[x], [y]$ and random triple $[a], [b], [c]$:

- To set the honest parties' $[x]$ for the rest of the simulation, recall that $\mathcal{S}$ knows the corrupted parties' $[x] = [e] + [a]$ via and their $[e]$ message and committed triple. Then, $\mathcal{S}$ computes $r' = p - \sum_U [x]$. Then for the rest of triples consumption, $\mathcal{S}$ sets the honest parties' $[x]$ to be an additive share of r', ie. $[x] = [r'] \bmod \tau_t$. Similarly for $[y]$, compute $\tilde{r}' = q - \sum_U [y]$ (where $[y] = [d] + [b]$) then set $[y] = [\tilde{r}'] \bmod \tau_t$ for the honest parties.
- Next, $\mathcal{S}$ updates the honest parties' triples values $[a], [b]$ to be consistent with the previously-sent messages. That is, $\mathcal{S}$ updates each honest party's $[a]$ to be $[r'] - [e] \bmod \tau_t$. Similarly, $\mathcal{S}$ sets the honest parties' $[b]$ to be $[\tilde{r}'] - [d] \bmod \tau_t$.
- Last, $\mathcal{S}$ computes the honest parties' last triples value: $[c]$. Since $\mathcal{S}$ computed the honest $[a], [b]$ in the previous step, it does $c' = a \cdot b - \sum_U [c] \bmod \tau_t$. Then each honest party is given an additive share of c', ie. $[c] = [c'] \bmod \tau_t$.
- $\mathcal{S}$ continues to the next stage of the simulation. Note that once $\mathcal{S}$ knows the prime factors p, q, it will always use them to simulate.

Emulating $\mathcal{F}_{\mathsf{ctriple}}$, $\mathcal{S}$ records updated triples $\{(\mathsf{assign}, sid, ssid, \mathcal{S}, [a], [b], [c])\}_{j\in[n]-U}$ for every honest party. Note that the simulation will abort with overwhelming probability in certification phase.

4. **Candidate generation.** $\mathcal{S}$ buckets the primes $d_{T_1+1}, \ldots, d_{T_2}$ into T' buckets and denote the products of the primes in the corresponding buckets $\overline{\tau}_1, \ldots \overline{\tau}_{T'}$. At index i^* (as defined in Step 3), if $\mathcal{S}$ knows the factors p, q of N, then the simulator follows Step 3b to set the honest parties' prime shares and random triple values but using $\overline{\tau}_1, \ldots \overline{\tau}_{T'}$ as moduli.

   Otherwise, the simulator follows Step 3a to emulate the honest parties for triples consumption, but using $\overline{\tau}_1, \ldots \overline{\tau}_{T'}$ as moduli and $N \bmod \overline{\tau}_t$ (where $t \in [T']$) as the target xy value. Note that if the adversary sends an incorrect message at any point (including for the $[e], [d]$ messages), $\mathcal{S}$ will set the honest shares and triple values according to Step 3b. Subsequently, $\mathcal{S}$ uses these values to complete the rest of the simulation according to the real protocol.

5. **Jacobi test.** For the execution corresponding with index i^*, $\mathcal{S}$ does the following s times:

   (a) $\mathcal{S}$ samples γ_{i^*} uniformly from the elements in $\mathbb{Z}_N^*$ with Jacobi symbol 1 according to the proof of [BF01]'s Lemma 4.2. In detail, $\mathcal{S}$ picks random $x \in \mathbb{Z}_N^*$ and $b \in \{0, 1\}$, and then sets $\gamma_{i^*} = x^2 \cdot (-1)^b \bmod N$.

   (b) If $\mathcal{S}$ knows the factors p, q of N, then the simulator computes and sends $\gamma_{i^*}^{(-p_{i^*,j}-q_{i^*,j})/4}$ to C on behalf of the honest parties such that $p_{i^*,j}, q_{i^*,j}$ are consistent with the honest parties' previously-generated shares. Specifically, compute $p_{i^*,j}, q_{i^*,j}$ via applying the CRT reconstruction algorithm to the honest $[x], [y]$ shares with respect to the moduli $\tau_1, \ldots, \tau_T$. Recall that these shares are set by the simulator in Step 3b if the adversary deviates in pre-sieving. Additionally (to set the honest shares for moduli when the adversary acts honestly), the simulator samples the honest shares as additive shares of

   $$p - \sum_{j\in U}[x] \bmod \tau_t = p - (\sum_{j\in U}[e] + \sum_{j\in U}[a]) \bmod \tau_t.$$

   (c) Otherwise if $\mathcal{S}$ does not know the factorization of N. Recall first that [BF01]'s Jacobi test checks whether $\gamma_{i^*}^{(N-p-q+1)/4} \equiv \pm 1 \bmod N$ where

   $$\gamma_{i^*}^{(N-p-q+1)/4} = \gamma_{i^*}^{(N+1)/4} \cdot \gamma_{i^*}^{-(\sum_{j\in U} p_{i^*,j} + \sum_{j\in U} q_{i^*,j})/4} \cdot \gamma_{i^*}^{-(\sum_{j\notin U} p_{i^*,j} + \sum_{j\notin U} q_{i^*,j})/4}$$

   and $\sum_{j\in U} p_{i^*,j}$ and $\sum_{j\in U} q_{i^*,j}$ are computed as the CRT reconstruction of the $\sum_U[x]$ and $\sum_U[y]$ values, respectively. Denoting by $u = \gamma_{i^*}^{-(\sum_{j\in U} p_{i^*,j} + \sum_{j\in U} q_{i^*,j})/4}$, then $\mathcal{S}$ assigns each honest party a multiplicative random share of c, with c satisfying

   $$\gamma_{i^*}^{(N+1)/4} \cdot u \cdot c \equiv b' \bmod N$$

   and $b' = (-1)^b$. Finally, each honest party sends its share of c to the coordinator C. This simulation step is based on the observation from [BF01] that the sign of b' is determined by whether γ_{i^*} is a quadratic residue mod N for γ values that have a Jacobi symbol 1. Therefore, setting $b' = (-1)^b$ properly simulates the protocol.

   Next, $\mathcal{S}$ receives the corrupted parties' corresponding messages and checks that they are consistent with the committed randomness and prior transcript. If $\mathcal{A}$ sends the messages honestly, the simulator simply continues to the next step in the simulation. If $\mathcal{A}$ uses randomness different

from what it was previously committed (i.e. deviates) for any corrupted party P_j, the simulator sends $(\mathsf{corrupt}, sid, ssid, P_j)$ to the functionality $\mathcal{F}_{\mathrm{RSA-ML}}$ and receives the prime factors p, q of N.

6. **GCD test.** For the execution corresponding with index i^*, $\mathcal{S}$ does the following:

   If $\mathcal{S}$ knows the factors p, q of N, it emulates the honest parties as follows:

   - First, if $\mathcal{S}$ hasn't set the honest parties' $p_{i^*,j}, q_{i^*,j}$ values, $\mathcal{S}$ emulates them according to the method from Step 5b. Next, on behalf of honest parties, $\mathcal{S}$ samples $a_j \in \mathbb{Z}_N$. Then for each bucket B_k, it samples $[e], [d] \bmod B_k$ for honest parties and receives the corrupted parties' $[e], [d]$ messages, thus setting the honest triples values. $\mathcal{S}$ simulates the rest of the GCD test honestly according to the real-world protocol.

   If $\mathcal{S}$ doesn't know the factors p, q of N,

   (a) In each bucket B_k, $\mathcal{S}$ emulates triples consumption for computing $z_k = a \cdot (p_i + q_i - 1) \bmod B_k$. $\mathcal{S}$ samples the honest parties' $[e], [d] \leftarrow [0, B_k - 1]$ and sends them to C. Following that, $\mathcal{S}$ receives the corrupted parties' $[e], [d]$. (The corrupted parties' $[d]$ messages are already determined by previous transcript, namely, $\mathcal{S}$ can reconstruct $p_{i,j}, q_{i,j}$ and has access to the corrupted parties' triple. Further, the corrupted $[e]$ message sets a_j. Thus, $\mathcal{S}$ can test that a_j (which is reconstructed in Q_{GCD}) is a value in $\mathbb{Z}_N^*$.) If a cheat is detected, it (emulating functionality $\mathcal{F}_{\mathsf{ctriple}}$) records $(\mathsf{abort}, sid, ssid, P_j)$ and sends $(\mathsf{corrupt}, sid, ssid, P_j)$ to the functionality $\mathcal{F}_{\mathrm{RSA-ML}}$ and receives the prime factors p, q of N.

   (b) Next, compute the honest parties' $[xy] = [z_k] \bmod B_k$ messages as described in Step 3a, replacing the target value $xy \bmod \tau_t$ with $z = a \cdot (p_i + q_i - 1) \bmod B_k$ where $a \in \mathbb{Z}_N^*$ and $(p_i + q_i - 1)$ is consistent wit the previous transcript. On behalf of honest parties, sample $v_j \in [\pm V]$. Then, for all k, compute $[\alpha_k]_j = [z_k] + v_j \cdot N \bmod B_k$. Once this is done in all buckets, use the CRT reconstruction algorithm to get $\alpha_j \bmod Q_{\mathrm{GCD}}$ and send α_j to C. Then, receive α_j from corrupted parties. $\mathcal{S}$ checks that the corrupted parties' α_j satisfy

      i. $N \mid (\alpha_j - a_j(p_{i,j} + q_{i,j}))$ where a_j and $p_{i,j} + q_{i,j}$ are consistent with previous transcript.
      ii. $\tilde{v}_j = N \mid (\alpha_j - a_j(p_{i,j} + q_{i,j}))$ is in $[\pm V]$.

      If these checks don't pass for any P_j, then $\mathcal{S}$ sends $(\mathsf{corrupt}, P_j)$ to the functionality $\mathcal{F}_{\mathrm{RSA-ML}}$ and receives the prime factors p, q of N.

7. **Certification.** To simulate certification, $\mathcal{S}$ emulates the role of $\mathcal{F}_{\mathsf{ctriple}}$ and receives the parties' certify messages. If $\mathcal{S}$ has previously sent an abort message for any party P_j, then it emulates sending an abort message from $\mathcal{F}_{\mathsf{ctriple}}$, sending $(\mathsf{abort}, sid, ssid, P_j)$ to C.

8. **Σ-protocol.** $\mathcal{S}$ invokes the simulator for the Σ-protocol proof system.

9. **Output.** Conditioned on the success of the the certification and Σ-protocol proof, $\mathcal{S}$ completes the simulation as described in the protocol.

Next, we argue security by proving indistinguishability of simulation via a sequence of intermediate hybrid distributions. We begin with the following cases.

Case 1: The adversary has deviated before reaching the pre-sieving phase. Note that in this case, the parties have not yet used their RSA product shares and therefore the only deviation by the adversary is abort which aborts the simulation as well. Therefore, the simulation is perfect.

Case 2: The adversary has deviated during or after the pre-sieving phase. Note that in this case, the simulator uses fake random shares on behalf of the honest parties, which are inconsistent with the RSA product N obtained from $\mathcal{F}_{\mathrm{RSA-ML}}$. Consequently, the differences between the real and the simulated executions are with respect to the messages produced by the honest parties.

Hybrid $\mathbf{H}_0$. In this hybrid experiment, we consider an execution that proceeds identically to the real world. The output of the experiment is defined as the output of the environment.

Hybrid $\mathbf{H}_1$. This hybrid is defined by a simulator $\mathcal{S}_1$ that plays the role of the honest parties as in the previous hybrid with the exception that it fakes the certification step as described in Step 7 of the simulator. Since the honest party always passes the certification in the real world, the output of this hybrid is identically distributed to the previous hybrid.

Hybrid $\mathbf{H}_2$. This hybrid is defined by a simulator $\mathcal{S}_2$ that plays the role of the honest parties as in the previous hybrid with the exception that it completes the biprimality **GCD test** step as described in Step 6 of the simulator.

- If the adversary does not deviate during this phase,

  The simulated honest $[e], [d]$ messages are sampled uniformly mod B_k for all k. The real protocol first selects triples values (ie masks) $[a], [b] \bmod B_k$ and adds them to $[x] = a_j \bmod B_k$ and $[y] = p_{i,j} + q_{i,j} \bmod B_k$ respectively. Using the CRT reconstruction algorithm to interpret all values mod Q_{GCD}, it easy to verify that these distributions differ information-theoretically since $\log(a_j) = 2\ell$ and $\log(p_{i,j} + q_{i,j}) = \ell$.

  Next, the honest parties' $[z_k]$ values are computed in relation to the corrupted parties' correct $[z_k]$ values, ie they are consistent with the corrupted parties' previous $[e], [d]$ messages and their triples. Thus, honest $[z_k]$ values additively share the expected honest party sum: $z - \sum_{j \in U} [z_k] \bmod B_k$. Hence, the $[z_k]$ values are identically distributed with in the real world. Then, then $\mathcal{S}$ samples v_j from the real distribution, so the honest α_j messages are distributed as in the real protocol.

- If the adversary deviates and the deviation occurs in the first message (i.e. corrupted parties' $[e], [d]$ messages), the simulator obtains the factorization of the candidate and then samples inputs and triples for honest parties consistent with the factorization and completes te GCD phase honestly. indistinguishability follows from the passive security of Beaver triples. Now, to analyze the case when the adversary deviates when sending the α_j message, observe that the honest party has no additional (passive protocol) messages, so this cannot influence the distribution of honest messages.

Hybrid $\mathbf{H}_3$. This hybrid is defined by a simulator $\mathcal{S}_3$ that plays the role of the honest parties as in the previous hybrid with the exception that it invokes the simulator of the Σ-protocol at the last phase. Indistinguishability here follows from the indistinguishability of the simulation of the Σ-protocol.

Hybrid $\mathbf{H}_4$. This hybrid is defined by a simulator $\mathcal{S}_4$ that plays the role of the honest parties as in the previous hybrid with the exception that it completes the Jacobi test as described in Step 5 of the simulator.

- If the adversary does not deviate during this phase, the simulation is perfect and it follows from Lemma 4.2 in [BF01].

- If the adversary deviates, first observe that the simulation perfectly simulates the real γ_i values due to Lemma 4.2 in [BF01]. Next, $\mathcal{S}$ samples the honest parties' $\gamma_i^{(-p_{i,j}-q_{i,j})/4}$ as random multiplicative shares in relation to the product of the corrupted parties' values, which are constructed as specified in the protocol. Thus, the distribution of the honest parties' messages are identical to in the real-world protocol. Hence, the broadcasted $\gamma_i^{(-p_{i,j}-q_{i,j})/4}$ also perfectly simulate the real distribution.

Hybrid $\mathbf{H}_5$. This hybrid is defined by a simulator $\mathcal{S}_5$ that plays the role of the honest parties as in the previous hybrid with the exception that it completes the Pre-Sieving phase as described in Step 3 of the simulator.

- If the adversary does not deviate during this phase, the real and simulated views are indistinguishable at index i^*. In the real protocol, the $[a]$, $[b]$, $[x]$, $[y]$ values are sampled as uniform, independent values mod τ_t. Then, computing $[e] = [x - a]$ and $[d] = [y - b]$ is distributed identically to sampling $[e], [d]$ uniformly from $[0, \tau_t - 1]$. Hence, sampling uniformly from $[0, \tau_t - 1]$ in the simulation perfectly emulates the honest parties' $[e], [d]$ messages. Next, since the simulated $[xy]$ are generated using the corrupted parties' real inputs and $xy = N \bmod \tau_t$, the honest parties' simulated $[xy]$ messages are identically distributed to the real messages.
- If the adversary deviates during this phase, we claim that the simulated and real views are indistinguishable. First, observe that in the simulation, the honest $[e], [d]$ messages are sampled uniformly mod τ_t, ie they do not depend on the triples received from $\mathcal{F}_{\mathsf{ctriple}}$. Hence, regardless of if the underlying triples values are real (received from $\mathcal{F}_{\mathsf{ctriple}}$) or set by the simulation, the distribution of these messages is identical. Thus, if $\mathcal{S}$ receives incorrect $[xy]$ values from any corrupted party, it can update the honest shares $[x], [y]$ and assign triple values $[a], [b], [c]$ upon learning the factors p, q, as described in the simulation's Step 3b.

Hybrid $\mathbf{H}_6$. This hybrid is defined by a simulator $\mathcal{S}_6$ that plays the role of the honest parties as in the previous hybrid with the exception that it completes the **Candidate generation** phase as described in Step 4 of the simulator.

- If the adversary does not deviate during this phase, this argument follows exactly from the previous hybrid, except with replacing moduli τ_t with $\overline{\tau}_t$.
- If the adversary deviates for the first time in this step, this argument follows exactly as in the previous hybrid.

Hybrid $\mathbf{H}_7$. This hybrid corresponds to the simulator defined above.

Case 3: The adversary has not deviated. Observe that this case is equivalent to following the *Case 2* hybrids when the adversary acts honestly. Indistinguishability follows accordingly.

■

5 Concrete Costs

5.1 The Number of Prime Candidates

In this section we analyze the number of primes candidates required for successfully generating a valid RSA composite (if all the parties were honest). We call running $\Pi_{\mathrm{RSA-ML}}$ from top to bottom once, regardless of if an output is produced, an iteration. We refer to the $p_{i,j}, q_{i,j}$ values output from the *CRT Reconstruction* step of $\Pi_{\mathrm{RSA-ML}}$ as prime candidates.

First, we need to understand the probability an ℓ-bit integer is prime, given that it survives trial division up to a bound B. Observe that this corresponds directly to prime candidates in $\Pi_{\mathrm{RSA-ML}}$ with trial division up to bound T_1. De Bruijn [Bru50] shows

$$\begin{aligned} p_{\mathsf{prime}} &= \Pr[p \text{ is prime} \mid \text{trial division up to } B] \\ &= 2.57\frac{\ln B}{\ell}\left(1 + o\left(\frac{1}{\ell}\right)\right). \end{aligned}$$

Then, the probability that the product of two such integers N_i is actually biprime is $(p_{\mathsf{prime}})^2$. If we try $\mathcal{N}'$ possible biprimes, then the probability that we fail to find any biprime in a single iteration is $(1-(p_{\mathsf{prime}})^2)^{\mathcal{N}'}$.

Now, we calculate the values above based on our parametrization. With $\ell = 1024$ and a trial division bound $B = 863$ (the 150th prime), we get that $p_{\mathsf{prime}} \geq 1/60$. Thus, any N_i produced by our protocol is a biprime with probability at least $1/3600$. This implies we want to generate at least 3600 possible biprimes, ie $\mathcal{N}' \geq 3600$. Last, given these prime candidates, the probability that we actually generate at least one biprime is $\approx 36.78\%$.

Number of CRT coefficient samples. In the *Pre-sieving* step from protocol $\Pi_{\mathrm{RSA-ML}}$, each α_t is the product of odd primes $d_{i_1}, d_{i_2}, \ldots d_{i_l}$. Then the probability a random sample $r_{i,t}^j \leftarrow [0, \alpha_t - 1]$ has $\mathrm{GCD}(r_{i,t}^j, \alpha_t) = 1$ is $p_{\mathsf{td}} = \prod_{x=1}^{l} (1 - (1/d_{i_x}))$. As we want $\mathcal{N}' = 3600$ biprime samples in expectation, we need the parties to sample $\lceil 2 \cdot 3600 \cdot \frac{1}{p_{\mathsf{td}}} \rceil$ times for each t.

Biprimality test: number of iterations. From the soundness of the biprimality tests, the Jacobi test has soundness $1/2$. Hence we need to repeat the Jacobi test s times to get soundness 2^{-s}. However, we only need to run the Jacobi test s times for the potential modulus. We therefore run it once for every candidate, then s more times for the survivors. In practice, we have observed that every modulus that passes the first Jacobi test is eventually a modulus. Similarly to the Miller-Rabin test [DLP93], we believe that the Boneh-Franklin tests indeed have stronger soundness for large moduli.

5.2 Communication Complexity

We calculate the total per-party communication for generating a 2048-bit modulus among 1024 parties. First, we show that all distributed multiplications can be packed into a single Ring-LWE element with a packing factor of $2^{16} = 65536$ (see Table 1 for all Ring-LWE parameters). Then we count the number of ring elements transferred for a single iteration of the protocol. Knowing that each ring element in R_Q is $(21 \cdot 8 \cdot 65536/(1024 \cdot 1024))$MB $= 10.5$MB, we arrive at our total communication estimate.

We show that a single Ring-LWE element with our parametrization gives sufficiently-many slots for all of the distributed multiplications in our protocol.

First consider the multiplications required for pre-sieving. Since our Ring-LWE parameters supports 175-bit modular arithmetic, we need $\lceil 1024/175 \rceil = 6$ buckets for generating 1024-bit primes via CRT. We use a greedy strategy to bucket the primes so that the probability of elimination in each bucket is roughly the same.

Since the probability of a sample failing in a bucket is roughly $1/p$ where p is the smallest prime in the bucket, the worst case is losing roughly half of the candidates in pre-sieving. Therefore, we require at least $2 \cdot 3600 \cdot 6 = 43200$ multiplications. Note that using only 43200 multiplications will leave empty packing slots in our Ring-LWE element, so our implementation allocates 43900 multiplications to maximize the number of prime candidates generated.

For the candidate generation step, we need to multiply two 1024-bit numbers using CRT. With each CRT factor around 175-bits, this requires 12 buckets. However, we already know the remainders of the modulus w.r.t the first 150 primes due to pre-sieving, so we do not recompute them. We only compute the product w.r.t the additional CRT factors whose product exceeds 1024-bits. Pre-sieving gives 3600 biprime candidates, so we require another $3600 * 6 = 21600$ products.

Finally for the GCD test, we need to multiply a 1024 bit number with a 2048 bit number, requiring 18 buckets. Overall, we need $43900 + 21600 + 18$ multiplications, which is less than 65536, the Ring-LWE packing factor.

We now compute the total communication cost. The key-generation and pre-sieving steps require transferring 14 ring elements overall, i.e. $14 \cdot 10.5 = 147$MB per party. The rest of the protocol amounts to less

than 3MB per iteration. To conclude, a single run of our protocol requires each party to transfer 150MB, split evenly between upload and download.

For active security, we estimate that our zero-knowledge proof will require less than 20MB per party. We provide more details in Section 6.3.

5.3 Round Complexity

We analyze the round complexity for our protocol. Key generation requires 2 rounds, and triples generation requires 3 rounds. Each of pre-sieving, candidate generation and GCD test requires 2 rounds. The Jacobi test runs twice and will incur 4 rounds of communication. This boils down for a total of 15 rounds of communication.

To reduce the number of rounds, we can perform the pre-sieving in parallel with the triples generation and the GCD test in parallel with the second iteration of the Jacobi test. This reduces the round complexity of our passive protocol to 11. For active security we include a commitment round at the beginning of the protocol and this brings our round complexity to 12.

6 Instantiating Our Primitives

6.1 Instantiating Our AHE Based on Ring-LWE

This section describes the Ring-LWE scheme used to implement multiplication in our protocol.

Notations. We emphasize that the notation in this section is self-contained. For an integer p, we write $\mathbb{Z}_p$ as the cyclic group $\{0, 1, \ldots, p-1\}$ with addition modulo p and $\mathbb{Z}[x]$ is the set of polynomials with integer coefficients. we write function $\exp(y)$ to mean e^y. For a vector $\boldsymbol{x}$, the Euclidean norm is $\|\boldsymbol{x}\|_2 = \sqrt{\sum_i |x_i|^2}$ and the ℓ-max norm is $\|\boldsymbol{x}\|_\infty = \max_i |x_i|$.

Polynomial rings. Our encryption scheme's plaintext and ciphertext spaces are polynomial rings of the form $R_P = \mathbb{Z}_P[X]/\Phi_m(X)$, where $\Phi_m(x) \in \mathbb{Z}[x]$ is the m^{th} cyclotomic polynomial with degree $n = \phi(m)$ (the totient function of m). For our scheme, we set $\Phi_m(x) = x^n + 1$ where m is a power of 2, so $n = m/2$. Recall that the ring R_P represents all polynomials up to degree $n-1$ with coefficients in $[0, P-1]$. We choose the modulus P to be a product of primes $p_1, \ldots, p_h$ such that $2n = 2^{k+1}$ divides $p_i - 1$. Then, for every i, there is a $2n$-th root of unity $\zeta_i \in \mathbb{Z}^*_{p_i}$. Let $\zeta = (\zeta_1, \ldots, \zeta_h)$ be the corresponding element in $\mathbb{Z}_P = \Pi_i \mathbb{Z}_{p_i}$.

The plaintext and ciphertext spaces are R_P and R_Q, respectively. Both P and Q will only have factors p_i satisfying $2n | p_i - 1$. We choose P and Q as follows. First, choose $Q = \Pi_{i=1}^h p_i$, where $p_1, \ldots, p_h$ are the first h small primes. Next, we specify that P is the product of a subset of the primes in Q, ie $P = \Pi_{i \subset [h]} p_i$.

For our implementation, we parametrize using values listed in Table 1.

Message packing. A message in our scheme is in $\mathbb{Z}_P$. We encode messages bigger than P via vectorizing the messages as $\boldsymbol{m} \in \mathbb{Z}_P^n$. For simplicity, we call $\mathbb{Z}_P^n$ our message space. Recall that the plaintext space of our encryption scheme is R_P.

To keep our communication lightweight, we pack an n-length message into a single polynomial in R_P. Intuitively, to map from $\boldsymbol{m} \in \mathbb{Z}_P^n$ to a polynomial $m(X) \in R_P$, we want the components of $\boldsymbol{m}$ to be the roots of unity for a polynomial $m(X) \in R_Q$, ie $\boldsymbol{m}_i = m(\zeta^{2i+1})$ for all $i \in [0, n-1]$. We compute $m(X)$ using the inverse of the discrete fast Fourier transform (FFT).This implicitly means that we need a polynomial of degree n to encode a message of length n to have sufficiently-many roots of unity. Note that recovering $\boldsymbol{m}$ simply involves applying FFT to $m(X)$.

We observe that FFT is a ring isomorphism that maps polynomial addition and multiplication (in R_P) to vector addition and pointwise multiplicaton in $\mathbb{Z}_P^n$. Hence, doing homomorphic operations is easy to

understand and implement. Note that to multiply by a scalar c, one must first map c to R_P using inverse FFT, then multiply.

Threshold homomorphic encryption from Ring-LWE. We describe a threshold additively homomoprhic encryption scheme supporting the computation of sums and affine functions based on the Ring-LWE problem. Using the notation from Definition 3, the protocol works as follows:

1. Secret key shares and randomness consists of triples $\mathrm{SK}_i = (a_i, s_i, e_i)$ where $a_i \in R_Q$ is chosen uniformly at random, and $s_i, e_i \leftarrow \chi$ are sampled from the LWE truncated discrete gaussian error distribution.

2. The public key reconstruction function takes as input n ring elements $x_i \in R_Q$ and outputs their sum.

$$\mathsf{Pub}(x_1, \ldots, x_n) = \sum_i \boldsymbol{x}_i \in R_Q$$

3. Key generation has two rounds, defined by the function

$$\begin{aligned} \mathsf{Gen}((a_i, s_i, e_i), []) &= a_i \in R_Q \\ \mathsf{Gen}((a_i, s_i, e_i), [a]) &= s_i \cdot a + e_i \in R_Q. \end{aligned}$$

   In the first round, each party sends a random ring element a_i and receives the sum $a = \sum_i a_i = \mathsf{Pub}(a_1, \ldots, a_n)$. In the second round, each party uses a (and its secret key) to compute the public share $b_i = s_i \cdot a + e_i$, and receives

$$b = \sum_i b_i = \mathsf{Pub}(b_1, \ldots, b_n) = s \cdot a + e$$

   where $s = \sum_i s_i$ and $e = \sum_i e_i$. The public key is $\mathrm{PK} = (a, b) \in R_Q^2$. Once the public key is computed, the values a_i and s_i are no longer needed, and the secret key can be simply set to $\mathrm{SK}_i = s_i$.

4. The (randomized) distributed decryption algorithm $\mathsf{Dec}(\mathrm{SK}_i, (c, d); r)$, on input a secret key share $s_i \in R_Q$ and ciphertext $(c, d) \in R_Q^2$, outputs $m_i = \delta_{i,1} \cdot d - s_i \cdot c + r$, where $r \in R_Q$ is chosen at random from a sufficiently large interval $[-U, +U]$ described below, and $\delta_{i,1}$ equals 1 if $i = 1$ and 0 otherwise.

5. The message space is R_P for some P dividing Q. The output reconstruction algorithm

$$\mathsf{Rec}(m_1, \ldots, m_n) = \left\lfloor (P/Q) \sum_i m_i \right\rceil \pmod P$$

   sums the message shares and rounds the (coefficients of the) sum to the closest multiple of Q/P.

Next, we describe the (randomized) encryption and homomorphic evaluation function Eval. The scheme supports the following functions:

1. Constant functions $f() \in R_P$, used to compute the encryption of a message $x \in R_P$. These are evaluated as a standard Ring-LWE encryption:

$$\begin{aligned} &\mathsf{Eval}(\mathrm{PK}, 0, f()) = \mathsf{Enc}((a, b), x; u, v, w) \\ &= \quad (a \cdot u + v, b \cdot u + w + (Q/P)x) \end{aligned}$$

2. Sums $f_\Sigma(x_1, \ldots, x_n) = \sum_i x_i \in R_P$. These are evaluated deterministically as the sum of the corresponding ciphertexts:

$$\mathsf{Eval}(\mathrm{PK}, n, f_\Sigma, [\boldsymbol{c}_1, \ldots, \boldsymbol{c}_n]) = \sum_i \boldsymbol{c}_i$$

3. Affine functions $f_{y,z}(x) = yx + z$, where $y, z \in R_P$ are ring elements, possibly from a restricted subset of R_P. The evaluation is randomized, and outputs

$$\begin{aligned}&\mathsf{Eval}(\mathrm{PK}, 1, f_{y,z}, [\boldsymbol{c}]; u, v, w)\\ &\quad = \quad y\boldsymbol{c} + (au + v, bu + w + (Q/P) \cdot z)\end{aligned}$$

where $u, v, w \leftarrow \chi$ as in the encryption queries

Parameters and the allowed sequences of queries are chosen to satisfy our correctness and security requirements. For any Ring-LWE ciphertext (c, d) encrypting a message m under key s, define the error

$$\mathsf{Err}_s((c, d), m) = d - sc - (Q/P)m.$$

At any point during the evaluation of a sequence of queries one can define an upper bound on the error of the ciphertexts $|\mathsf{Err}_s(\boldsymbol{c}_q, m_q)| \leq \beta_q$. These bounds depend on the sequence of operations in the scheme. These bounds depend on the sequence of operations supported by the scheme or required by the application. See Appendix B for bounds specific to our protocol.

- For security, we require the size U of the error added by decryption queries q to be bigger than β_q by a factor 2^κ exponentially large in the security parameter κ.
- For correctness, the modulus Q should be larger than $2PU\beta_q$, so that rounding eliminates the error, and recovers the correct message.

We emphasize that the amount of noise U added in distributed decryption operations should be tuned to the error bounds β_q specific to the sequence of operations performed by the protocol. This is similar to the common requirement in lattice-based (leveled) homomorphic encryption that the modulus Q be chosen large enough to correct the errors that arise from the execution of specific homomorphic operations. Once the bounds β_q have been determined, the correctness and security of our threshold homomorphic encryption scheme can be proved in a rather generic way, as shown in the next two theorems.

Theorem 4. *Assume $Q > 2P(nU + \beta_q)$ for all decryption queries q. Then, the Ring-LWE threshold homomorphic encryption scheme is correct.*

Proof. The error in the decryption queries is bounded by β_q by definition. Each party in the distributed decryption algorithm adds at most U to the error. So, before the rounding operation performed by Rec, we have

$$\sum_i m_i = \frac{Q}{P}m + r$$

where m is the correct output message and r is an error term bounded by $\beta_q + nU < Q/(2P)$. Multiplying by (P/Q) and rounding to the closest integer, Rec correctly recovers $m \pmod P$. ∎

Theorem 5. *Assume $U > 2^\kappa \beta_q$ for all decryption queries q. Then, the Ring-LWE threshold homomorphic encryption scheme is secure under the standard hardness of Decisional Ring-LWE.*

Proof. Consider a security game as described in Definition 5 with challenge bit $\dot{b}$, and assume without loss of generality that the adversary picks the index $h = 1$. We make a sequence of modifications to the security game (formally, we define a sequence of hybrids) that alter the adversary's advantage only by a negligible amount. The changes lead to a game where the adversary has advantage 0. It follows that the advantage in the original game from the definion must be negligible.

Let $\mathrm{SK}_i = (a_i, s_i, e_i)$ be the secret key shares picked at random for $i = h = 1$ or semi-maliciously by the adversary for $i \geq 2$, and define $a = \sum_i a_i$, $\bar{a} = \sum_{i \geq 2} a_i = a - a_1$, and similarly for $s, \bar{s}, e, \bar{e}$ and $b, \bar{b}$.

Notice that since the adversary has to pick the a_i's (and their sum $\bar{a}$) before seeing a_1, and a_1 is chosen uniformly at random, the value $a = a_1 + \bar{a}$ is also uniformly distributed. Since the a_i values are not used anywhere else, we can replace $a \in R_Q$ with a uniformly random ring element, and ignore the a_i's.

Next we look at the decryption queries. Let (c_q, d_q) be the qth (decryption) query, and m_q the associated message. Notice that in the security game this message does not depend on the challenge bit because of the constraint $m_q^0 = m_q^1$. So, this message can be efficiently computed by the adversary or by a simulator. The decryption query is answered with the honest party partial decryption

$$\begin{aligned} d_q - s_1 \cdot c_q + r_q &= d_q - (s - \bar{s}) \cdot c_q + r_q \\ &= \quad (Q/P)m_q + r'_q + \bar{s} \cdot c_q + r_q \end{aligned}$$

where $r'_q = \mathsf{Err}_s((c_q, d_q), m_q)$, by definition, is bounded by $\|r'_q\|_\infty < \beta_q$, and r_q is chosen uniformly at random from $[-U, +U]$. Since $U \geq 2^\kappa \beta_q \geq 2^\kappa \|r'_q\|_\infty$, the distribution of $r_q + r'_q$ is statistically close (within distance $2^{-\kappa}$) to that of r_q, without the addition of r'_q. So, we can replace the decryption oracle answer with a value

$$(Q/P)m_q + \bar{s} \cdot c_q + r_q$$

which can be computed by the adversary on its own because it knows both m_q and $\bar{s}$. In particular, the decryption query is answered without using the honest key share s_1.

After repeating the above process for all decryption queries, we see that the honest secret key share s_1 is not used anywhere, except for the initial computation of the public $b_1 = a \cdot s_1 + e_1$, during the key generation stage. Under the Ring-LWE assumption, this value is computationally indistinguishable from a uniformly random $b_1 \in R_q$. So, we can replace b_1 with a truly random values. Moreover, similarly to a, since the adversary has to choose all e_i, s_i (and the resulting $b_i = as_i + e_i$ and $\bar{b}$) before receiving any information about b_1, the sum $b = b_1 + \bar{b}$ is also uniformly random.

At this point we can also replace b with a uniformly random ring element, and ignore all other values computed during the key generation stage. So, both a and b are now uniformly random and independent.

At this point, both a and b are uniformly random and independent. To conclude, we consider the challenge queries, which are the only queries that depend on the hidden bit $\dot{b}$ of the security game. Our cryptosystem supports three types of evaluation queries: encryption, sums, and affine functions. These queries are answered as follows:

- Sum queries are trivial because for every n there is only one sum function $f_\Sigma(x_1, \ldots, x_n) = \sum_i x_i$. So, the adversary can compute $f_0 = f_1 = f_\Sigma$ on its own, without knowing the challenge bit $\dot{b}$.

- Encryption queries (m_0, m_1) are answered with

  $$\mathsf{Enc}(\mathrm{PK}, m_{\dot{b}}) = (au + v, \dot{b}u + w) + (0, m_{\dot{b}} \cdot Q/P).$$

  Since a, b are uniformly random, and u, v, w are chosen from the LWE error distribution χ, the pair $(a, au + v)$ and $(b, bu + w)$ are LWE samples with secret u and noise v, w respectively. So, under the Decisional Ring-LWE assumption, $au + v$ and $bu + w$ are indistinguishable from uniformly random values. So, we can answer the query with

  $$\mathsf{Enc}(\mathrm{PK}, m_{\dot{b}}) = (x, y) + (0, m_{\dot{b}} \cdot Q/P)$$

Parameter	Notation	Value
Security parameter	κ	128
Number of parties	N	1024
Gaussian parameter	σ	8
Degree/Packing Factor	n	2^{16}
Ciphertext Modulus Size	$\|Q\|$	1302 bits
Plaintext Modulus Size	$\|P\|$	558 bits
Maximum number of bits for τ	$\mathsf{max\_bits}(\tau)$	175 bits

Table 1: Ring-LWE choice of parameters.

where x, y are chosen uniformly at random and independently from a, b. Adding $(0, m_{\dot{b}} \cdot Q/P)$ maps the uniform distribution to itself. So, we can also eliminate $(0, m_{\dot{b}} \cdot Q/P)$, and answer the encryption query $\mathsf{Enc}(\mathsf{PK}, m_{\dot{b}})$ with a pair of uniformly random ring elements $(x, y) \in R_Q^2$, independently of the bit $\dot{b}$.

- Affine queries $(y_0, z_0), (y_1, z_1)$ are treated similarly. Just as before, the answer to the query

$$\mathsf{Eval}(\mathsf{PK}, 1, f_{y_{\dot{b}}, z_{\dot{b}}}, [\boldsymbol{c}]; u, v, w) = (y_{\dot{b}}\boldsymbol{c} + (0, z_{\dot{b}}Q/P)) + (au + v, bu + w)$$

is the sum of a fixed value $(y_{\dot{b}}\boldsymbol{c} + (0, z_{\dot{b}}Q/P))$ and a pair $(au + v, bu + w)$ which is indistinguishable from a uniformly random element of R_Q^2. So, we can answer the query with just a pair of random ring elements $(x, y) \in R_Q$.

At this point all queries are answered without using the challenge bit $\dot{b}$ at all. So, the adversary advantage in guessing the value of $\dot{b}$ is 0. ■

Implementation parameters. Our code implements Ring-LWE operations using the open source library NFLlib [nfl], with parameters given in Table 1. Recall that both P and Q have prime factors p_i such that $2n \mid p_i - 1$ for message-packing purposes. Then with $n = 2^{16}$, we have the constraint $2^{17} | p_i - 1$. Hence, each p_i needs to be $\geq 2^{18}$ to allow for a sufficient message packing.

Our implementation sets p_i to be 62 bits for better soundness with respect to the zero-knowledge proofs. In detail, the statistical soundness is the inverse of the smallest prime factor's field size, so we get soundness 2^{-62}. NFLlib provides the mechanisms to easily force each factor of P and Q to be 62 bits: one simply specifies the desired number factors for P and Q. We set $P = 9$ and $Q = 21$.

Lastly, based on the noise analysis for our multiplication protocol (Appendix C), the moduli P and Q are related via the inequality $Q > 2^{\kappa+2}\sigma P^2 N^2 n$. It is easy to verify that our parameterization satisfies this inequality.

6.2 Identifying Prime Buckets for Triples Generation

During the triples generation phase, party P_j samples vectors $\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c_j} \in [B_1] \times [B_2] \times \cdots \times [B_n]$. Recall that the triples are used in three places: (1) Pre-sieving, (2) Candidate generation, and (4) GCD steps. We discuss here how the B_i's are chosen.

For pre-sieving, we need to first identify the prime buckets $B_1, \ldots, B_T$. Recall that the product of B_i's must equal the product of the first T_1 primes excluding 2, and each B_i is at most 175-bits given our

instantiation of the underlying TAHE scheme. Let $B = p_1 \times p_2 \times \cdots \times p_W$. Using the inclusion-exclusion princple, we have that the probability a random number mod B_i is not relatively prime with B_i is

$$\sum_i 1/p_i - \sum_{i,j} 1/p_i p_j + \sum_{i,j,k} 1/p_i p_j p_k - ...$$

where all the summation are over distinct indices. Denote this probability be q_{B_i}. We use a greedy strategy to bucket the primes across the B_i's so $\max_i q_{B_i}$ is minimized. Then to choose the number of triples needed for each B_i, we proportionally divide the total number of triples available. Namely, we allocate $1/(1-q_{B_i})$ fraction of the triples for B_i.

For candidate generation, we allocate enough triples to multiply pairs of 2048-bit numbers that survive the pre-sieving phase. We multiply by deconstructing the numbers into CRT factors of at most 175-bits and reconstructing after multiplying across the individual CRT moduli. Since the product of the shares modulo $B_1, \ldots, B_T$ is already available from the pre-sieving phase, we already have 1024 bits of information. We just need enough additional moduli for the remaining 1024 bits. We choose 175-bit primes directly for these moduli.

For the GCD test, we multiply two numbers modulo $Q = (3 \times 1024 + 2 \times \log n + 128 + 5)$ bits. We do this only once for the candidate that survives the Jacobi test.

In order to divide the total number of triples n (i.e. the degree of the encryption scheme) between the three categories, first we allocate enough for the GCD phase. Since we need roughly 3600 candidates to get one bi-prime in expectation, next we allocate enough triples for 3600 multiplications in the candidate generate phase. Finally, we distribute the remaining for the pre-sieving as described above. For $n = 65536$, we averaged between 3000 and 3600 candidates from the pre-sieving using our allocation. Let $n_{\mathsf{PS}}, n_{\mathsf{CAN}}, n_{\mathsf{GCD}}$ be the number of triples allocated for each of the three categories respectively.

6.3 Implementing $\mathcal{F}_{\text{CP}}$ (Commit-and-Proof)

We will implement the commit-and-prove functionality using the Ligero zero-knowledge argument system [AHIV17] and Shoup's Σ-protocol [Sho00]. We first describe our NP statement and then discuss the implementation. We list the components of the NP statement:

1. **Key generation:** Recall that the parties commit to $s_i, e_i \in R_Q$ at the beginning of the protocol, where s_i and e_i are elements of $\mathbb{Z}_Q^n$. Then, party P_j needs to prove that:
   (a) Each element of the vectors s_j and e_j will be in the range $[-10\sigma, 10\sigma]$ except with very small probability. $10\sigma = 80$ for our parameters.
   (b) Second, the b_j value transmitted by party P_j in the second round of the key generation protocol satisfies $b_j = a \times s_j + e_j \in R_Q$.
2. **Triples generation:** Parties choose the i^{th} elements of $\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j$ from B_i where $B_1, \ldots, B_n$ We use a greedy strategy to identify $B_1, \ldots, B_n$ and the number of triples we allocate for each category as described in Section 6.2. In this segment, the party proves that there exists $\boldsymbol{a}_j, \boldsymbol{b_j}, \boldsymbol{c_j} \in B_1 \ldots B_n$ and randomness such that:
   (a) The i^{th} elements of $\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j$ are in $\{0, 1, \ldots, B_i - 1\}$ and i^{th} element of $\boldsymbol{z}_j$ is in $[-nB_i 2^\lambda, nB_i 2^\lambda]$.
   (b) Each element of u_1, u_2 is in $[-80, 80]$.
   (c) Each element of v_1, w_1, v_2, w_2 is in $[-80, 80]$.
   (d) Party P_j transmits $\alpha_j = (a \cdot u_1 + v_1, b \cdot u_1 + w_1 + (Q/P)m_1) \in R_Q^2$ to the coordinator in Step 1 of triples generation.

(e) m_1 is a polynomial over R_P such that $m_1(\zeta_i^f) = a_j[f] \bmod p_i$ for $i \in \{1, \ldots, 9\}$ and $1 \leq f \leq 65536$.

(f) $\beta_j = \mathsf{Eval}(\mathsf{PK}, 1, f_{lin}(\boldsymbol{b}_j, \boldsymbol{c}'_j, \cdot), \alpha) = (m_2\beta_1 + a \cdot u_2 + v_2, \beta_2 + b \cdot u_2 + w_2 + (Q/P)m_3)$, where $m_2(\zeta_i^f) = b_j[f]$, $m_2(\zeta_i^f) = z_j[f] - c_j[f] \bmod p_i$ for $i \in \{1, \ldots, 9\}$ and $1 \leq f \leq 65536$.

(g) $d_j = \mathsf{Dec}(j, \mathsf{SK}_j, \beta) = \delta_i \cdot \beta_2 - \beta_1 s_i + r$, where $r \in R_Q$.

(h) Every entry in r is in $[-U, U]$.

3. **Pre-sieving:** The parties sample random shares and multiply using the triples generated in the previous phase. Each party P_j proves that there exists $r^j_{i,t}, \tilde{r}^j_{i,t}$ such that the triples were used correctly in the multiplication protocol using triples to compute $(\sum_i r^j_{i,t}) \times (\sum_i r^j_{i,t}) \bmod B_t$ and that $0 < r^j_{i,t}, \tilde{r}^j_{i,t} < B_t$.

4. **Candidate generation:** The parties prune their lists based on the result of the pre-sieving. Party P_j identifies prime shares $p_{i,j}$ and $q_{i,j}$. Next, it consumes triples to compute $N_i = (\sum_j p_{i,j}) \times (\sum_j q_{i,j})$. Recall that the parties already know $N_i \bmod B_t$ for all $t \in [T]$ from pre-sieving. In this step we compute $N_i \bmod B_{t+1} \ldots B_{T'}$ such that $\lfloor \log_2(\prod_{i=1}^{T'} B_i) \rfloor > 2048$. Parties prove that each candidate $p_{i,j}$ is the reconstructed value from pre-sieving and that the triples were consumed correctly.

5. **Jacobi test:** The parties raise γ to the exponent $(-p_{i,j} - q_{i,j})/4$. The parties employ the Σ-protocol proof of knowledge for an exponent of an unknown order group w.r.t. γ [Sho00].

6. **GCD test:** The parties prove that the triples were consumed correctly w.r.t. $p_{i,j}, q_{i,j}$ in a similar way to candidate generation.

We introduce the following optimization that reduces the size of the NP statement. Upon concluding the protocol execution, the parties reveal all of the triples except those used to generate the survived candidates. This means parties only need to prove correctness w.r.t triples that led to these biprimes.

The first issue with this optimization is that we rely on packed ciphertexts for the Ring-LWE encryption scheme. We want to keep some elements in the encrypted vectors private while revealing others. Interestingly, our version of packing can be "unpacked". Given $a, s, e, b \in R_Q$ such that $as + e = b$, we can extract $A[i] \in \mathbb{Z}_Q^n$ such that $\langle A[i], s \rangle + e[i] = b[i]$. This insight to simplifies the equations in Items 2d and 2f. More precisely, by unpacking these equations, it suffices to bound specific indices of the vector v_1, w_1, v_2, w_2 for Item 2c and all elements of u_1 and u_2. Suppose we need to keep f triples private. In fact, if we want to keep f positions of $\boldsymbol{a}_j, \boldsymbol{b}_j$ and $\boldsymbol{c}_j$ hidden, we can bound the first f elements of the vectors v_1, w_1, v_2, w_2. Recall that the LWE encryption scheme encrypts polynomials m_1, m_2, m_3 and the actual messages are the evaluation of these polynomials on specific roots of unity. Therefore, to hide f evaluations of m_1, m_2, m_3 it suffices to keep any f coefficients of the polynomial private and reveal the remaining in the clear. For the vectors in Items 2a and 2h, it suffices to bound the specific indices that contribute to the final biprime.

We will incorporate our NP statement using three proof systems that will share witnesses. Then, in order to compose the proofs, we argue that the three proof systems have identical values for the shared portions of their witnesses. The three proof systems are:

1. **Range proofs.** We will use the Ligero zero-knowledge system to prove the constraints in Items 1a and 2b, where the underlying field is $\mathbb{F}_{p_1}$.

   We follow a standard approach to provide range proofs. In order to prove a comparison $v < L$, the prover includes the bit representation of $v = v_0 v_1 \cdots v_j$ and another number $v' = v'_0 v'_1 \cdots v'_j$ in the witness. It proves that: (1) both v_i and v'_i are bits using a quadratic constraint $x^2 - x = 0$, and (2) $\sum_{i=0}^{j-1} 2^i \cdot (v_i + v'_i) + (2^j - L) = 2^j$ where $j = \lceil \log_2(L) \rceil$, and (3) all computations are modulo p_1.

2. **Σ-protocol.** We rely on a standard discrete log Σ-protocol [Sho00] to prove correctness in the Jacobi test (see Figure 6). We note that it suffices to prove correctness of the Jacobi test for the final biprime as the shares for the remaining candidates are revealed in the clear.

3. **Main proof.** We provide a second proof using the Ligero zero-knowledge system that incorporates the rest of the NP statements listed above. Recall that our LWE ciphertext modulus Q is the product of 21 different 61-bit primes $p_1, \ldots, p_{21}$. We generate a proof of the remaining constraints modulo each of these primes.

Proving consistency between the three proof systems. In order to compose the three different proof systems, we identify the overlapping parts of the witnesses.

In order to compose the three different proof systems we need to argue that the overlapping witnesses are identical in the different proof systems. First, we identify the overlapping parts of the witness.

1. **Range proof and main proof:** The values s_i, e_i, u_1, u_2 are part of the witness in the range proof and main proof. Hence, we need to show that the same values for each of the variables have been incorporated in the witnesses of both proofs. First we remark that we use the first of the 21 moduli to perform the range proof. The main proof on the other hand factors each of the equations into the 21 moduli by taking all equations/variables modulo the corresponding prime. Hence, the first moduli incorporates part of the main proof as well as the whole of the range proof. In order to show that the main proof and the range proof incorporates the same values for the overlapping variables, we prove two things: (1) First, we prove that each element of the vector s_i (and similarly for all the other values) that $s_i \bmod p_1$ is between -80 and 80 using bit-decomposition. Observe that since $80 << p_1$ we have that $s_i \bmod p_i$ must be s_i (for an honest party). (2) We prove that $s_i \bmod p_1$ is equal to $s_i \bmod p_2 \cdots p_{21}$. For this we first choose r_{blind} uniformly at random from $[p_1]$ that we incorporate in the witness for the first moduli p_1 and in the rest of the moduli via CRT decomposition. Then, we introduce a linear function that computes an inner product over $\mathbb{F}_{p_1}$ with inputs (a) the vector obtained by combining all the elements from s_i, e_i, u_1, u_2 and r_{blind} and (b) another random vector of the same length where each element is chosen uniformly at random from $[p_1]$. We compute the output of this linear function from the first moduli and the from the rest and compare the values. As the inner product is computed moduli p_1, to compute this over $p_2 \cdots p_{21}$ we include a large multiple of p_1 to the inner product so that only the value $\bmod p_1$ is revealed. Finally, we remark that that the random vector in (b) above is obtained via Fiat-Shamir.

2. **Σ-protocol and main proof:** The overlapping parts of the two witnesses are the shares of the primes $p_{i,j}, q_{i,j}$. To show that the sigma protocol used the right values, we will prove that the $p_{i,j}, q_{i,j}$ values in the main proof satisfies the linear constraint $z = r + ex$, which is revealed in the third step of the sigma protocol. For soundness, the witness must be committed in advance. This holds since we commit $\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j$ at the beginning.

   Our current active implementation does not include the stitching of the proofs together. We remark that our results will not be significantly affected after we incorporate this part. In particular, the computational and communication cost involves adding one linear constraint for each of the two cases and will result in a small additive overhead (< 5MB) in the generation of the zero-knowledge proofs.

PROTOCOL Π_{DL}

Inputs. $u, v \in \mathbb{Z}_N^*$. A private input for the prover $\mathcal{P}$ is x such that $x = \log_v u$.

Prover's first message. The prover chooses a random value r of length $2\ell + s$ bits (where 2ℓ is the modulus length and s is the statistical security parameter), and sends $a = v^r \bmod N$.

The verifier's challenge message. The verifier $\mathcal{V}$ responses with a random challenge message e of length t bits.

Prover's third message. $\mathcal{P}$ sends its third message $z = r + ex$ (over the integers) and $\mathcal{V}$ checks that $v^z = au^e$.

Figure 6: Σ-protocol for proving the knowledge of a discrete log in groups with unknown order.

7 Implementation and Experiments

We developed a robust and optimized implementation of our protocol in approximately 14500 lines of C++ code (excluding external libraries) with an additional 3300 lines of unit, integration, and end-to-end testing code. Our implementation's networking layer defines primitives for sending messages and awaiting replies and a separte encryption protocol performs the computation and tests the data. A substantial effort was made to properly select Ring-LWE, commitment, zero-knowledge, and hashing parameters in order to achieve roughly 128-bits of security.

Software engineering. We use ZeroMQ [zer], a fast request-reply concurrency framework for all networking operations. We used the ZMQ_ROUTER pattern for the coordinator, which is a bidirectional unrestricted pattern that checks the identities of the parties to determine where to route messages. This pattern drops messages once a high watermark backlog is reached for all peers, which is called a *mute state*. For our parties, we use the ZMQ_DEALER pattern, which is also bidirectional. But, instead of dropping messages, it blocks once a party gets to the mute state. We manually tuned parameters for the parties and the coordinator to increase the high watermark for our coordinator to 15,000 messages. We configure a Keep-Alive strategy to lower network load.

NFLlib enhancements. We implement ring operations using NFLlib [nfl], a highly optimized, open source C++ library for ideal lattice cryptography. We changed NFLlib's default allocator to use the heap instead of the stack in order to support our scheme's polynomial degree parameters which exceeded the defaults.

Coordinator optimizations. Our protocol coordinator performs two types of operations: data aggregation and broadcasting the result of the aggregation. To do data aggregation, coordinator receives data shares from the parties, deserializes and validates them, and then applies a given operation in a way similar to map-reduce. We do operations over vectors, including summation and multiplication defined as the Hadamard (element-wise) product. Data aggregation is much more computationally intensive than broadcasting, so we optimized those operations by running the coordinator runs on multiple CPUs with substantial amount of RAM and implementing a Divide-and-Conquer approach that splits the data into blocks, performs a given operation on these blocks, and promotes results to the top of the call chain. In contrast, our optimizations for the party code are algorithmic only and assume single-threaded execution.

Distributed verifier. For active security, our coordinator verifies all parties' public data and proofs. To parallelize and expedite verification, we introduce multiple verifiers who run as separate processes. The coordinator receives the parties' proofs and load-balances them to the verifiers. Once the verification process is done, the verifiers send reports back to coordinator. If all reports succeed, the coordinator finishes the ceremony.

In active mode our coordinator verifies parties public data and proofs. In order to speed up this process we parallelize it accross multiple verifiers running them as separate processes. The coordinator receives

proofs from parties and redirects them to the corresponding verifier without deserialization acting as a load balancer in this case. Each verifier can handle data for all modulis and for one or a few parties. Along with proofs coordinator sends public data based on shares submitted by parties. Once verification process is done all verifiers send the verification reports back to coordinator. If all reports are successful coordinator finally finishes the ceremony.

Serialization. After analyzing running times and profile information, one of the main bottlenecks of our protocol was data serialization and deserialization on the coordinator side. To lower network load, we used Boost binary serialization to minimize the amount of data transferred.

One of the major improvements for serialization that we implemented was multithreaded aggregation of data on the coordinator side. Since the coordinator runs on a powerful node, we implemented a multithreaded deserialization for the coordinator, i.e. any aggregation first splits data into chunks then uses an approach similar to Divide-and-Conquer to redistribute deserialization among multiple threads.

Throughput test. The total amount of time the coordinator spends on sending or receiving messages depends not only on the coordinator's hardware or network, but also on the parties' network quality. At the very beginning of our protocol, even before parties register to participate, we perform a throughput test for each party wanting to join. Using ZeroMQ primitives, this throughput test measures the amount of time needed to transfer a given amount of data to the party and to receive an acknowledgement message from it. We implement the test with ZeroMQ primitives.

During the throughput test, each party pushes as much data as it can during a fixed time frame. If a party pushes enough data, it passes the test. In our experiments, the throughput test cuts off all parties with an uplink speed slower than 1Mbps. Only parties that survived the throughput test are allowed to register and join the protocol computation. Of course, these parameters should be tuned for various cases; for cases with a slow network but increasing the total time of the protocol is allowable, increasing the uplink speed threshold allows slower parties to participate. Meanwhile for extremely large experiments (≥ 1000 parties) or for benchmarking the protocol, we used nodes from data centers distributed far from each other, thus inducing a relatively poor network. For this case, turning off the throughput test is necessary to prevent eliminating most parties.

In more detail, the throughput test begins when the coordinator sends a message to every party client. In response to the coordinator's message, each party sends $\texttt{nb\_max\_send} = 1024$ packages, each of $\texttt{data_size} = 32$kB, to the coordinator. Then, based on the number of packages received within a $\texttt{duration}$ of 20secs, the coordinator calculates the party's network bandwidth. If a party has less than the $\texttt{threshold\_cutoff}$ of 0.8Mbps bandwidth, it is eliminated. Next, to mitigate against denial-of-service events, we set the following time bounds on the throughput test. A party will time out in the following cases: 1) The party waits longer than $\texttt{wait_timeout} = 5$mins for the coordinator to begin the test, and 2) After the throughput test terminates, the party fails to receive a stop message from the coordinator within $\texttt{timeout} = 25$secs.

On the coordinator side, we set the following time bounds. The coordinator waits a $\texttt{wait\_time}$ of 2mins for parties to join the throughput test. After the actual throughput test, the coordinator must cleanup the queue (of lagging packages). If no package arrives within an interval of $\texttt{cleanup_time} = 5$secs, the cleanup is done. Additionally, the coordinator times out after 1min.

Protocol restarts. Our protocol may restart if no RSA biprime is sampled or if it identifies a cheating party. While the protocol may restart arbitrarily-many times, our implementation restarts at most 10 times. If no RSA biprime is sampled, observe that no party is kicked out. As soon as all candidates are eliminated, the protocol restarts at triples generation (and skips running another throughput test).

The coordinator identifies cheating parties in the certification/zero-knowledge phase. If any party fails to send a proof (including when the party times out) or if any party sends an incorrect proof, the protocol kicks out that party and restarts at triples generation with the remaining parties. If any (distributed) verifier

Node Type	Location	AWS EC2 Region
Parties	N. Virginia	`us-east-1`
Parties	Ohio	`us-east-2`
Parties	N. California	`us-west-1`
Parties and Coordinator	Oregon	`us-west-2`

Table 2: AWS EC2 Data Center Locations.

aborts during the certification and zero knowledge because of poor network connection or faulty node, we simply record the party's proof on an AWS instance. Then, the parties' proofs can be verified at a later time.

7.1 Experiments

7.1.1 Setup

We prepared a Docker image file that runs on Ubuntu 18.04 LTS with kernel version 4.9.184 and the set of libraries we use: Boost 1.69, GMP 6.1.2, etc. For AWS, we created an AMI which matches our Docker image and then provisioned this image to all of our party nodes. Using our test orchestration harness (Section 7.1.1), anyone can easily reproduce our results. Our code has been made available at `github.com/ligeroinc/LigeroRSA`.

To simulate the real-world scenario, we executed our experiments on the AWS Cloud platform using `t3.small` nodes for each party; these nodes run on a shared 2.5 Ghz Intel Xeon with 2 virtual CPUs and 2GB of RAM. Our process is shared with other tasks running on the AWS cloud hardware, and AWS provides up to 5Gbps network performance for these nodes. Our coordinator ran on `r5dn.24xlarge` nodes with 96 virtual CPUs, 768GB of RAM, and up to 100 Gbps network. In all of our experiments, the coordinator node is always in Oregon (`us-west-2`). For tests with $\leq$ 1000 parties, party nodes were distributed in N. Virginia (`us-east-1`) and Ohio (`us-east-2`) AWS EC2 regions. For tests with $>$ 1000 parties, party nodes were distributed in N. Virginia (`us-east-1`), Ohio (`us-east-2`), Oregon (`us-west-2`), and N. California (`us-west-1`) regions.

Test orchestration harness. In order to perform experiments and gather logs on hundreds to thousands of nodes, we developed an orchestration harness on top of AWS tools and basic command line tools. This harness helps us build and deploy binaries to the nodes, validate them after deployment, and finally launch them seamlessly at scale. After computation all the results are uploaded to AWS S3 storage and could be easily fetched from there for further data analysis. The whole harness consists of a dozen of script tools or about 600 lines in Bash. While some of these scripts depend on AWS tooling, others are independent from the type of cloud used, ie the whole harness can be easily ported to another type of the cloud.

7.1.2 Empirical Results

Table 3, we present empirical data on the runs of our protocol. For $n \leq 1000$ parties, we run our protocol at least 5 times and report averages for all metrics; party-side metrics are averaged across all n parties as well. For larger instances $n > 1000$, we report the result of a single run and the party-side metrics are computed as an average over a subset of 1000 randomly sampled parties. When reporting timings, we measure the wall-clock time as measured from either the coordinator, party, or distributed verifier depending on the metric, to produce one modulus in expectation[4].

[4]Note, since this functionality is inherently a sampling functionality, there is a noticeable failure probability to produce a modulus. This probability can be adjusted by setting parameters; we set the parameters to produce one answer in expectation.

Parties	**Passive** ($\mu \pm \sigma$ **s**)	**Active** ($\mu \pm \sigma$ **s**)	**Registration (s)**	**# Runs** (passive/active)
2	23.8 ± 2.6	431 ± 26.9	26.3	30/18
5	22.7 ± 1.8	408 ± 25.3	25.7	29/13
10	25.0 ± 2.6	445 ± 25.3	33.4	29/14
20	27.1 ± 4.9	444 ± 6.7	40.4	30/17
50	39.6 ± 7.4	463 ± 5.2	40.0	12/4
100	56.1 ± 6.8	486 ± 10.8	31.4	30/18
200	99 ± 19.9	590 ± 9.5	28.7	5/3
500	137 ± 5.3	853	81.6	8/1
1000	211 ± 2.5	1306	126.3	6/1
2000	366 ± 8.3	2550	119.6	3/1
4046	684 –	4581	158.7	1/1

Table 3: Overall running time of protocol to sample a 2048-bit modulus in expectation with 128-bit security. Timings are averages over the number of runs along with standard deviations when applicable.

We report the *registration* time for the protocol separately. This is the time it takes for all n nodes to register with the coordinator and perform a throughput test to ensure that the node has enough bandwidth to complete the protocol. This phase is orthogonal to the protocol.

Memory We instrumented the client nodes to measure their peak memory usage. As expected, this value does not change substantially as n increases. We measured usage from 1857.59–1861.91 MB of RAM usage as n varied from 2 to 1000.

However, memory usage on the coordinator is indeed a bottleneck in our scaling to larger parties. In our current implementation of the active-secure protocol, the coordinator in each round stores each message sent by each of the parties because the NP statement that the party gives is with respect to this message. Later, to verify the proof, the coordinator will need to send this message to the distributed verifier. This implementation detail limits us to running 4000 parties because we hit RAM limitations on our coordinator instance.

Protocol communication. In our coordinator model, each participant's communication complexity is independent of n. Table 4 summarizes the size of each major message in the protocol. The "Gather Proofs" row reports an average of the total of the 21 proofs that are sent, and 17 other small messages are combined in the "All Others" row. The most expensive message, as predicted, is the gathering of public data which consists of sending the NP statement to the distributed verifiers.

Protocol timing. Our protocol requires 12 rounds of communication to complete the five basic steps of Key generation, Beaver triple generation, Modulus construction, Pre-Sieving, and the biprimality testing. We instrumented the protocol to measure how long each step takes as the number of parties increases. As expected, the triple generation step requires the most amount of time.

Active security Our protocol uses zero-knowledge proofs of honest behavior to achieve active security. In particular, there are 21 statements for which each node provides a ZK proof, the coordinator arranges these proofs and sends them to distributed verifiers. As noted in Table 3, the active security portion of the protocol dominates the running time. In this section, we further analyze the components of this step.

Figure 8 presents the average amount of time required by the party to produce each of the zero knowledge proofs as 21 segments per bar, the total time of the active protocol, and finally the verifier work. In particular,

Message Type	Size (b)
Public Key A	11,010,105
Public Key B	11,010,105
Encrypted X	22,020,166
Encrypted $XY + Z$	22,020,166
Sieving flags	175,724
AX Value	2,625,131
Modulus Candidate	1,901,993
All others	3,674
Total Semihonest	70,767,064
Gather Public Data	143,084,762
Gather Proofs (21)	35,106,330
Total Active	248,958,156

Table 4: Per Party communication complexity of the protocol

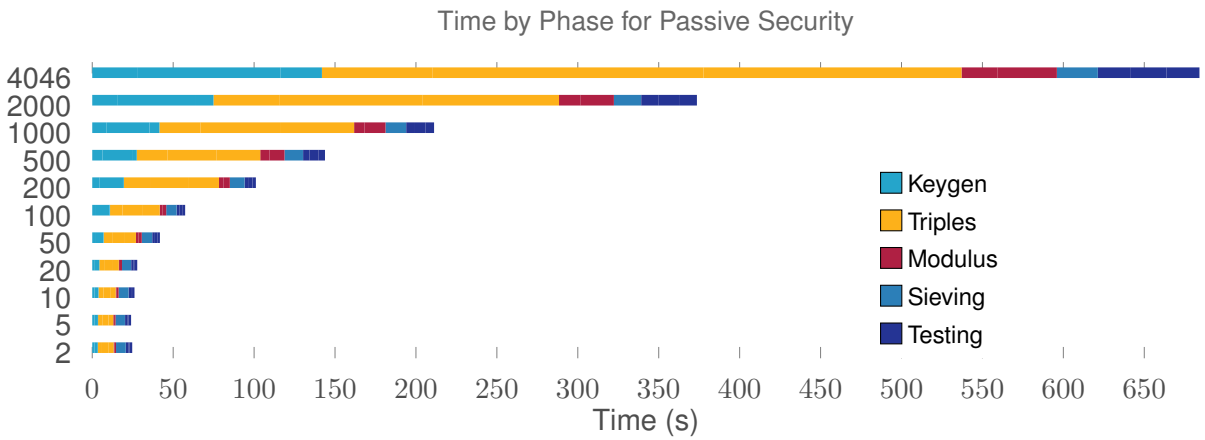

Figure 7: Cumulative timing per step of the protocol as the number of parties increases

the verifiers are often idle and waiting for messages—these times are depicted as blank gaps between the bars in the graph. Notice that the verifier work schedule remains fairly constant as n increases. This is expected because each verifier only considers one proof in our experiment. Neither of these work profiles account for the overall running time, which is dominated by the coordinator's task of shuttling messages. This graph suggests system-level improvements to the running time.

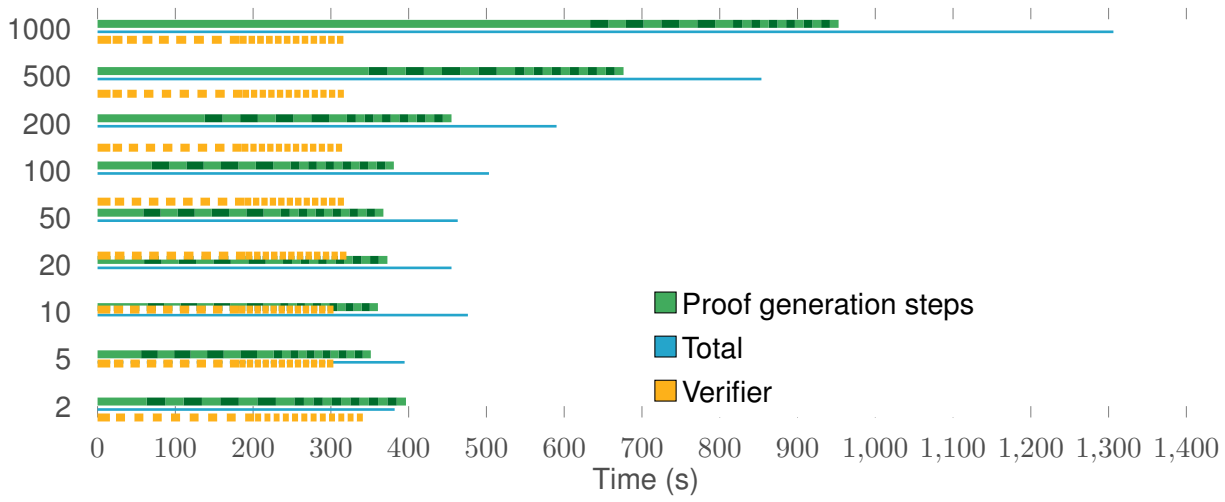

Figure 8: Cumulative timing per step of the active protocol as the number of parties increases.

8 Acknowledgements

We thank the Ethereum Foundation, Protocol Labs and the VDF Alliance for funding this project. We specifically thank Justin Drake, Dankrad Feist, Kelly Olson and Simon Peffers for giving us feedback throughout the development and relaying real-world concerns in deployment. We thank Nick Thompson for developing the initial transport architecture. We thank Matt DiBiase and Scott Catlin for their encouragement and logistical support.

References

[ACS02] Joy Algesheimer, Jan Camenisch, and Victor Shoup. Efficient computation modulo a shared secret with application to the generation of shared safe-prime products. In *CRYPTO*, pages 417–432, 2002.

[ADI+17] Benny Applebaum, Ivan Damgård, Yuval Ishai, Michael Nielsen, and Lior Zichron. Secure arithmetic computation with constant computational overhead. In *CRYPTO*, pages 223–254, 2017.

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *CCS*, pages 2087–2104, 2017.

[AJL+12] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In *EUROCRYPT*, pages 483–501, 2012.

[AO12] Gilad Asharov and Claudio Orlandi. Calling out cheaters: Covert security with public verifiability. In *ASIACRYPT*, pages 681–698, 2012.

[BBBF18] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *CRYPTO*, pages 757–788, 2018.

[BBBG99] Simon R. Blackburn, Simon Blake-Wilson, Mike Burmester, and Steven D. Galbraith. Weaknesses in shared RSA key generation protocols. In *Cryptography and Coding, 7th IMA International Conference, Cirencester, UK, December 20-22, 1999, Proceedings*, pages 300–306, 1999.

[BBC+19] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. In *CRYPTO*, pages 67–97, 2019.

[BBHR19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In *CRYPTO*, pages 701–732, 2019.

[BCP15] Elette Boyle, Kai-Min Chung, and Rafael Pass. Large-scale secure computation: Multi-party computation for (parallel) RAM programs. In *CRYPTO*, pages 742–762, 2015.

[BCR+19] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for R1CS. In *EUROCRYPT*, pages 103–128, 2019.

[BD10] Rikke Bendlin and Ivan Damgård. Threshold decryption and zero-knowledge proofs for lattice-based cryptosystems. In *TCC*, pages 201–218, 2010.

[BDO14] Carsten Baum, Ivan Damgård, and Claudio Orlandi. Publicly auditable secure multi-party computation. In *SCN*, pages 175–196, 2014.

[Bea91] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, pages 420–432, 1991.

[BF97] Dan Boneh and Matthew K. Franklin. Efficient generation of shared RSA keys (extended abstract). In *CRYPTO*, pages 425–439, 1997.

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

[BFO12] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In *CRYPTO*, pages 663–680, 2012.

[BFS19] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent snarks from DARK compilers. *IACR Cryptology ePrint Archive*, 2019:1229, 2019.

[BGG+18] Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter M. R. Rasmussen, and Amit Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. In *CRYPTO*, pages 565–596, 2018.

[BH08] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure MPC with linear communication complexity. In *TCC*, pages 213–230, 2008.

[BOS16] Carsten Baum, Emmanuela Orsini, and Peter Scholl. Efficient secure multiparty computation with identifiable abort. In *TCC*, pages 461–490, 2016.

[Bru50] N. G. De Bruijn. On the number of uncancelled elements in the sieve of eratosthenes. *Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen: Series A: Mathematical Sciences*, 53(5-6):803–812, 1950.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[CCD+20] Megan Chen, Ran Cohen, Jack Doerner, Yashvanth Kondi, Eysa Lee, Schuyler Rosefield, and abhi shelat. Multiparty generation of an RSA modulus. *Manuscript*, 2020.

[CDI+19] Melissa Chase, Yevgeniy Dodis, Yuval Ishai, Daniel Kraschewski, Tianren Liu, Rafail Ostrovsky, and Vinod Vaikuntanathan. Reusable non-interactive secure computation. In *CRYPTO*, pages 462–488, 2019.

[CDN01] R. Cramer, I. Damgård, and J. B. Nielsen. Multiparty computation from threshold homomorphic encryption. In *EUROCRYPT*, pages 280–299, 2001.

[Cle86] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *STOC*, pages 364–369, 1986.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. pages 494–503, 2002.

[Coc97] Clifford C. Cocks. Split knowledge generation of RSA parameters. In *Cryptography and Coding, 6th IMA International Conference, Cirencester, UK, December 17-19, 1997, Proceedings*, pages 89–95, 1997.

[Cop97] Don Coppersmith. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. *J. Cryptology*, 10(4):233–260, 1997.

[Des94] Yvo Desmedt. Threshold cryptography. *European Transactions on Telecommunications*, 5(4):449–458, 1994.

[DI06] Ivan Damgård and Yuval Ishai. Scalable secure multiparty computation. In *CRYPTO*, pages 501–520, 2006.

[DIK10] Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In *EUROCRYPT*, pages 445–465, 2010.

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS*, pages 1–18, 2013.

[DKLS19] Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. Threshold ECDSA from ECDSA assumptions: The multiparty case. In *IEEE Symposium on Security and Privacy, SP*, pages 1051–1066, 2019.

[DLP93] Ivan Damgård, Peter Landrock, and Carl Pomerance. Average case error estimates for the strong probability prime test. *Journal of Mathematics of Computation*, 61:177–194, July 1993.

[DM10] I. Damgård and G. L. Mikkelsen. Efficient, robust and constant-round distributed RSA key generation. In *TCC*, pages 183–200, 2010.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO*, pages 572–590, 2007.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, pages 643–662, 2012.

[EFKP19] Naomi Ephraim, Cody Freitag, Ilan Komargodski, and Rafael Pass. Continuous verifiable delay functions. *IACR Cryptology ePrint Archive*, 2019:619, 2019.

[FFS88] U. Feige, A. Fiat, and A. Shamir. Zero-knowledge proofs of identity. *J. Cryptology*, 1(2):77–94, 1988.

[FH96] Matthew K. Franklin and Stuart Haber. Joint encryption and message-efficient secure computation. *J. Cryptology*, 9(4):217–232, 1996.

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

[FMY98] Yair Frankel, Philip D. MacKenzie, and Moti Yung. Robust efficient distributed rsa-key generation. In *STOC*, pages 663–672, 1998.

[FS86] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, pages 186–194, 1986.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *EUROCRYPT*, pages 626–645, 2013.

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

[GLS19] Vipul Goyal, Yanyi Liu, and Yifan Song. Communication-efficient unconditional MPC with guaranteed output delivery. In *CRYPTO*, pages 85–114, 2019.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC*, pages 218–229, 1987.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT*, pages 305–326, 2016.

[HIMV19] Carmit Hazay, Yuval Ishai, Antonio Marcedone, and Muthuramakrishnan Venkitasubramaniam. LevioSA: Lightweight secure arithmetic computation. In *To appear CCS*, 2019.

[HIV17] Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Actively secure garbled circuits with constant communication overhead in the plain model. In *TCC*, pages 3–39, 2017.

[HMRT12] Carmit Hazay, Gert Læssøe Mikkelsen, Tal Rabin, and Tomas Toft. Efficient RSA key generation and threshold paillier in the two-party setting. In *CT-RSA*, pages 313–331, 2012.

[IKO+11] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, Manoj Prabhakaran, and Amit Sahai. Efficient non-interactive secure computation. In *EUROCRYPT*, pages 406–425, 2011.

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

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In *CRYPTO*, pages 572–591, 2008.

[JL09] S. Jarecki and X. Liu. Efficient oblivious pseudorandom function with applications to adaptive ot and secure computation of set intersection. In *TCC*, pages 577–594, 2009.

[JS07] S. Jarecki and V. Shmatikov. Efficient two-party secure computation on committed inputs. In *EUROCRYPT*, pages 97–114, 2007.

[Kil88] Joe Kilian. Founding cryptography on oblivious transfer. In *STOC*, pages 20–31, 1988.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014.

[KLR06] Eyal Kushilevitz, Yehuda Lindell, and Tal Rabin. Information-theoretically secure protocols and security under composition. In *STOC*, pages 109–118, 2006.

[KMTZ13] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In *TCC*, pages 477–498, 2013.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 830–842, 2016.

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT*, pages 158–189, 2018.

[LPR13a] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *J. ACM*, 60(6):43:1–43:35, 2013.

[LPR13b] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. A toolkit for ring-LWE cryptography. In *EUROCRYPT*, pages 35–54, 2013.

[nfl] NFLlib. https://github.com/quarkslab/NFLlib/.

[NS10] Takashi Nishide and Kouichi Sakurai. Distributed paillier cryptosystem without trusted dealer. In *WISA*, pages 44–60, 2010.

[Pai99] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, pages 223–238, 1999.

[Pie19] Krzysztof Pietrzak. Simple verifiable delay functions. In *ITCS*, pages 60:1–60:15, 2019.

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

[Rab98] Tal Rabin. A simplified approach to threshold and proactive RSA. In *CRYPTO*, pages 89–104, 1998.

[RST+19] Dragos Rotaru, Nigel P. Smart, Titouan Tanguy, Frederik Vercauteren, and Tim Wood. Actively secure setup for SPDZ. *IACR Cryptology ePrint Archive*, 2019:1300, 2019.

[RSW96] Ronald R. Rivest, Adi Shamir, and David A. Wagner. Time-lock puzzles and timed-release crypto. 1996.

[SF16] Gabriele Spini and Serge Fehr. Cheater detection in SPDZ multiparty computation. In *ICITS*, pages 151–176, 2016.

[Sho00] Victor Shoup. Practical threshold signatures. In *EUROCRYPT*, pages 207–220, 2000.

[Wes19] Benjamin Wesolowski. Efficient verifiable delay functions. In *EUROCRYPT*, pages 379–407, 2019.

[WRK17] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *CCS*, pages 39–56, 2017.

[XZZ+19] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *CRYPTO*, pages 733–764, 2019.

[zer] ZeroMQ. https://zeromq.org.

A UC Security

We briefly recall the Universal Composability (UC) framework, for more details we refer to [Can01].

Environment. The model of execution includes a special entity called the UC-environment (or environment) $\mathcal{Z}$. The environment "manages" the whole execution: it invokes all the parties at the beginning of the execution, generates all inputs and reads all outputs, and finally produces an output for the whole concurrent execution. Intuitively, the environment models the "larger world" in which the concurrent execution takes place (e.g., for a distributed computing task over the Internet, the environment models all the other activities occurring on the Internet at the same time).

Adversarial behavior. The model of execution also includes a special entity called the adversary, that represents adversarial activities that are directly aimed at the protocol execution under consideration. We consider *static* adversaries, where corruptions occur at the onset of the protocol execution. When a party is corrupted, it shares all its tapes with the adversary and follows its instructions for all its future actions.

While honest parties only communicate with the environment through the input/output of the functions they compute, the adversary is also able to exchange messages with the environment in an arbitrary way throughout the computation.[5] Furthermore, the adversary controls the scheduling of the delivery of all messages exchanged between parties (where messages sent by the environment are directly delivered). Technically, this is modeled by letting the adversary read the outgoing message tapes of all parties and decide whether or not and when (if at all) to deliver the message to the recipient, therefore the communication is asynchronous and lossy. However, the adversary cannot insert messages and claim arbitrary sender identity. In other words, the communication is authenticated.

We argue security of our protocol in the asynchronous network model where there is not guaranteed termination. However, we remark that our protocol proceeds in a lock-step manner, i.e. a message in round $i + 1$ will not be sent until all messages in round i have been sent. Hence, it is possible to express our protocol in a synchronous communication model using standard techniques [CLOS02, KLR06, KMTZ13].

Protocol execution. The *execution of a protocol π with the environment $\mathcal{Z}$, adversary $\mathcal{A}$ and trusted party $\mathcal{G}$* proceeds as follows. The environment is the first entity activated in the execution, who then activates the adversary, and invokes other honest parties. At the time an honest party is invoked, the environment assigns it a unique identifier and inquires the adversary whether it wants to corrupt the party or not. To start an execution of the protocol π, the environment initiates a *protocol execution session*, identified by a session identifier sid, and activates all the participants in that session. An activated honest party starts executing the protocol π thereafter and has access to the trusted party $\mathcal{G}$. We remark that in the UC model the environment only initiates one protocol execution session.

[5] Through its interaction with the environment, the adversary is also able to influence the inputs to honest parties indirectly.

Invoking parties. The environment invokes an honest party by passing input (invoke, P_i) to it. P_i is the globally unique identity for the party and is picked dynamically by the environment at the time it is invoked. Immediately after that, the environment notifies the adversary of the invocation of P_i by sending the message (invoke, P_i) to it, who can then choose to corrupt this party by replying (corrupt, P_i). Note that here as the adversary is static, parties are corrupted only when they are "born" (invoked).

Session initiation. To start an execution of protocol π, the environment selects a subset U of parties that has been invoked so far. For each party $P_i \in U$, the environment activates P_i by sending a start-session message (start-session, P_i, sid, $c_{i,sid}$, $x_{i,sid}$) to it, where sid is a session id that identifies this execution. We remark that in the UC model, the environment starts only one session, and hence all the activated parties have the same session id.

Honest party execution. An honest party P_i, upon receiving (start-session, P_i, sid, $c_{i,sid}$, $x_{i,sid}$), starts executing its code $c_{i,sid}$ input $x_{i,sid}$. During the execution,

- The environment can read P_i's output tape and at any time may pass additional inputs to P_i;
- According to its code, P_i can send messages (delivered by the adversary) to the other parties in the session, in the format (P_i, P_j, s, content),[6] where P_j is the identity of the receiver;
- According to its code, P_i can send an input to the trusted party in the format (P_i, $\mathcal{F}$, s, input).

Adversary execution. Upon activation, the adversary may perform one of the following activities at any time during the execution.

- The adversary can read the outgoing communication tapes of all honest parties and decides to deliver some of the messages.
- The adversary can exchange arbitrary messages with the environment.
- The adversary can read the inputs, outputs and incoming messages of a corrupted party, and instruct the corrupted party for any action.
- The (adaptive) adversary can decide to corrupt any party from the set of honest parties at the moment.

Output. The environment outputs a final result for the whole execution in the end.

In the execution of protocol π with security parameter $n \in \mathbb{N}$, environment $\mathcal{Z}$, adversary $\mathcal{A}$ and trusted party $\mathcal{G}$, we define $\text{REAL}^{\mathcal{G}}_{\pi,\mathcal{A},\mathcal{Z}}(n)$ to be the random variable describing the output of the environment $\mathcal{Z}$, resulting from the execution of the above procedure.

Let $\mathcal{F}$ be an ideal functionality; we denote by π_{IDEAL} the protocol accessing $\mathcal{F}$, called as the ideal protocol. In π_{IDEAL} parties simply interacts with $\mathcal{F}$ with their private inputs, and receive their corresponding outputs from the functionality at the end of the computation. Then the *ideal model execution* of the functionality $\mathcal{F}$ is just the execution of the ideal protocol π_{IDEAL} with environment $\mathcal{Z}$, adversary $\mathcal{A}'$ and trusted party $\mathcal{F}$. The output of the execution is thus $\text{IDEAL}^{\mathcal{F}}_{\pi_{\text{IDEAL}},\mathcal{A}',\mathcal{Z}}(n)$. On the other hand, the real model execution does not require the aid of any trusted party. Let π be a multi-party protocol implementing $\mathcal{F}$. Then, the *real model execution* of π is the execution of π with security parameter n, environment $\mathcal{Z}$ and adversary $\mathcal{A}$, whose output is the random variable $\text{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(n)$. Additionally, the *$\mathcal{G}$-Hybrid model execution* of a protocol π is the execution of π with security parameter n, environment $\mathcal{Z}$ and adversary $\mathcal{A}$ and ideal functionality $\mathcal{G}$.

[6] The session id in the messages enables the receiver to correctly de-multiplexing a message to its corresponding session, even though the receiver may involve in multiple sessions simultaneously.

Security as emulation of a real model execution in the ideal model. Loosely speaking, a protocol *securely realizes* an ideal functionality if it *securely emulates* the ideal protocol π_{IDEAL}. This is formulated by saying that for every adversary $\mathcal{A}$ in the real model, there exists an adversary $\mathcal{A}'$ (a.k.a. *simulator*) in the ideal model, such that no environment $\mathcal{Z}$ can tell apart if it is interacting with $\mathcal{A}$ and parties running the protocol, or $\mathcal{A}'$ and parties running the ideal protocol π_{IDEAL}.

Definition 7. (UC security) *Let $\mathcal{F}$ and π_{IDEAL} be defined as above, and π be a multi-party protocol in the $\mathcal{G}$-hybrid model. Then protocol π UC realizes $\mathcal{F}$ with static (resp. adaptive) security in $\mathcal{G}$-hybrid model, if for every uniform* PPT *static (resp. adaptive) adversary $\mathcal{A}$, there exists a uniform* PPT*simulator $\mathcal{A}'$, such that for every non-uniform* PPT *environment $\mathcal{Z}$, the following two ensembles are computationally indistinguishable,*

$$\left\{\text{REAL}^{\mathcal{G}}_{\pi,\mathcal{A},\mathcal{Z}}(n)\right\}_{n\in\mathbb{N}} \overset{\text{c}}{\approx} \left\{\text{IDEAL}^{\mathcal{F}}_{\pi_{\text{IDEAL}},\mathcal{A}',\mathcal{Z}}(n)\right\}_{n\in\mathbb{N}}.$$

Multi-session extension of ideal functionalities. Note that the UC model only considers a single session of the protocol execution. (Namely, the environment is only allowed to open one session). To consider multiple concurrent executions, we focus on the multi-session extension of ideal functionalities [Can01, CLOS02]. More specifically, let $\hat{\mathcal{F}}$ be the multi-session extension of $\mathcal{F}$. That is, $\hat{\mathcal{F}}$ runs multiple copies of $\mathcal{F}$, where each copy will be identified by a special "sub-session identifier". Every k parties, trying access $\mathcal{F}$ together, share a sub-session identifier, $ssid$. To compute the function, each party simply sends its private input together with $ssid$ to $\hat{\mathcal{F}}$. Upon receiving all the inputs, $\hat{\mathcal{F}}$ activates the appropriate copy of $\mathcal{F}$ identified by $ssid$ (running within $\hat{\mathcal{F}}$), and forwards the incoming messages to that copy. (If no such copy of $\mathcal{F}$ exists then a new copy is invoked and is given that ssid.) Outputs that are generated by the copies of $\mathcal{F}$ are returned to corresponding parties by $\hat{\mathcal{F}}$.

B Evaluating Ring-LWE Noise

We provide our analysis for the Ring-LWE noise of our parametrization (Section 6.1). We first present some lemmas useful for our parameterization. Subsequently, we determine the size of plaintexts, then analyze the noise for ciphertexts and homomorphic operations.

Lemma B.1. *The sum of N elements, each sampled from a discrete Gaussian distribution with parameter σ and mean 0 induces a random variable sampled from a discrete Gaussian distribution with parameter $\sqrt{N}\sigma$ and mean 0.*

Proof. It's well known from probability theory that the sum of independent random variables that are normally distributed is also normally distributed. Formally, let X_0, X_1 be independent, normally-distributed random variables both with respective means μ_0, μ_1 and parameters σ_0, σ_1. Then their sum $Y = X_0 + X_1$ has mean $\mu_0 + \mu_1$ and parameter $\sqrt{\sigma_0^2 + \sigma_1^2}$.

For our statement, we first extend this to be a sum of N random variables, first define the sum $Y = X_0 + \ldots + X_N$. Then Y will have mean $\sum_{j=1}^{N} \mu_j$ and parameter $\sqrt{\sum_{j=1}^{N} \sigma_j^2}$. Now, plugging in the original mean $\mu = 0$ and the Gaussian parameter $\sigma_j = \sigma$, we get that Y has mean 0 and parameter $\sqrt{N}\sigma$. ■

Lemma B.1 implies the following corollary.

Corollary 6. *The Euclidean norm of χ, a centered (0-mean) n-dimension Gaussian random variable with parameter $\sqrt{N}\sigma$ is at most $\sqrt{Nn}\sigma$.*

Next, we require the following lemma, which can be proved using the Chernoff bound.

Lemma B.2. *The sum of N elements, each sampled from a discrete uniform distribution over $[\pm R]$ (with $R \in \mathbb{Z}$) is at most $R\sqrt{N}$ except with negligible probability.*

Plaintext size. The shares x_i, y_i are sampled modulo τ. From Lemma B.2, Sampling x_i, y_i in the centered range $[\pm(\tau-1)/2]$ implies $x = \sum_{i=1}^{N} x_i$ and $y = \sum_{i=1}^{N} y_i$ are both at most $\tau\sqrt{N}$ with high probability, implying $x \cdot y \leq \tau^2 N$.

Our plaintext is $x \cdot y + \tau z$, with $z = \sum_{i=1}^{N} z_i$ and each $z_i \in [\pm Z]^n$. We first determine an appropriate bound for τz. We want $x \cdot y + \tau z$ to only depend on $xy \bmod \tau$ within a statistical distance of $2^{-\kappa}$. So, we require that $\tau z = 2^\kappa \max(|xy|) \leq 2^\kappa \tau^2 N$. Thus,

$$x \cdot y + \tau z \leq \tau^2 N + 2^\kappa \tau^2 N < 2^{\kappa+1} \tau^2 N.$$

So the plaintext has size at most $2^{\kappa+1}\tau^2 N$.

Now, we determine appropriate values for Z. From the above equation about τz, we deduce that $z \leq 2^\kappa \tau N$, so $\max(z) = 2^\kappa \tau N$. Our protocol defines $z = \sum_{i=1}^{N} z_i$ with each $z_i \in [\pm Z]^n$. From Lemma B.2, we have $z \leq Z\sqrt{N}$ with high probability. Note that we want $Z\sqrt{N} \leq \max(z)$ to ensure τz doesn't overflow. This means $Z\sqrt{N} \leq 2^\kappa \tau N$. So $Z \leq 2^\kappa \tau \sqrt{N}$. For our implementation, we set $Z = 2^\kappa \tau \sqrt{N}$.

Ciphertext noise. For a given ring element $x \in R_Q$, the absolute value $|x|$ is the ℓ-max norm of x's coefficients, which are centered in the interval $[\pm(Q-1)/2]$. Now, we define the error of an element of R_Q.

Definition 8 (Error of an element in R_Q)**.** *Let $((a,b),s) \leftarrow \mathsf{Gen}$ and $(c,d) \leftarrow \mathsf{Enc}_{(a,b)}(m)$. A ciphertext $(c,d) \in R_Q^2$ encrypts a message $m \in \mathbb{Z}_P^n$ with error β if $d - s \cdot c = (Q/P)m + e'$ and $|e'| < \beta$.*

We can compute the error of a fresh ciphertext as follows: First recall that the encryption function is

$$\mathsf{Enc}_{(a,b)}(m) = (ua + v, ub + w + (Q/P)m),$$

where u, v, w, e have coefficients sampled from χ. Then, setting $e' = eu + w - sv$, we can prove that $|e'| \leq 2\sigma^2\sqrt{Nn}$ for a fresh ciphertext as follows.

$$\begin{aligned} d - s \cdot c &= (ub + w + (Q/P)m) - s(ua + v) \\ &= ub + w + (Q/P)m - sua - sv \\ &= u(as + e) + w + (Q/P)m - sua - sv \\ &= sua + eu + w + (Q/P)m - sua - sv \\ &= eu + w + (Q/P)m - sv \\ &= (Q/P)m + eu + w - sv \end{aligned}$$

So $e' = eu + w - sv$. Then,

$$
\begin{aligned}
|e'| &= |eu + w - sv| \\
&= \|eu + w - sv\|_\infty \\
&\le \|eu + w - sv\|_2 \\
&= \sqrt{|eu|^2 + |w|^2 + |-sv|^2} \\
&= \sqrt{\|eu\|_\infty^2 + \|w\|_\infty^2 + \|-sv\|_\infty^2} \\
&\le \sqrt{(\|e\|_\infty \|u\|_\infty)^2 + \|w\|_\infty^2 + (\|-s\|_\infty \|v\|_\infty)^2} \\
&= \sqrt{(\|e\|_\infty \sigma)^2 + \sigma^2 + (\|-s\|_\infty \sigma)^2} \text{ (definition max-norm, } u, y \text{ sampled from } \chi_\sigma) \\
&= \sigma\sqrt{(\|e\|_\infty)^2 + 1 + (\|-s\|_\infty)^2} \\
&< \sigma\sqrt{2(\sigma\sqrt{Nn})^2 + 1} \text{ (Corollary 6)} \\
&< \sigma\sqrt{2(\sigma\sqrt{Nn})^2 + 1 + (2(\sigma\sqrt{Nn})^2 - 1)} \\
&= \sigma\sqrt{4(\sigma^2 Nn)} \\
&= 2\sigma^2\sqrt{Nn}
\end{aligned}
$$

Thus, the error of a fresh ciphertext is $2\sigma^2\sqrt{Nn}$. Now, we analyze the error of ciphertexts under homomorphic operations.

Addition. Since the error of ciphertexts is defined via upper bounds, the error of a sum of two ciphertexts is upper-bounded by the sum of their errors. In other words, $\sum_j C_j$ has error $\sum_j \beta_j$. However, if we have N fresh ciphertexts, each with error β, we can achieve a stronger noise bound of $\sqrt{N}\beta$ with high probability.

Scalar multiplication. Correspondingly, multiplying a ciphertext by α scales the error by α. In the worst case, this increases the error from β to $Pn\beta$. This can be reduced to $P\sqrt{n}\beta$ with high probability by randomizing the constant α in $[\pm P]$. (Lemma B.2).

C Noise Accounting for $\Pi_{\text{MULT-RLWE}}$

We determine the noise constraints required for the correctness of $\Pi_{\text{MULT-RLWE}}$. First, we determine the ciphertext noise before decryption.

- Each fresh ciphertext $\mathsf{Enc}(x_j)$ has noise
$$2\sigma^2\sqrt{Nn}.$$
- The sum of N independent ciphertexts $\mathsf{Enc}(x) = \sum_j \mathsf{Enc}(x_j)$ has noise
$$\sqrt{N}(2\sigma^2\sqrt{Nn}) = 2\sigma^2 N\sqrt{n}.$$
- Multiplying $\mathsf{Enc}(x)$ by a constant y_j (randomized in $[\pm P]$) scales the noise by $P\sqrt{n}$, so we get
$$P\sqrt{n}(2\sigma^2 N\sqrt{n}) = 2\sigma^2 PNn.$$

- Adding a fresh encryption of τz_j increases the noise only by a small additive term, which can be neglected.
- Summing N ciphertexts scales the noise by $\sqrt{N}$. This gives the ciphertext noise before decryption:

$$\sqrt{N}(2\sigma^2 PNn) = 2\sigma^2 PN^{1.5}n.$$

Now, we examine how the decryption protocol Π_{DEC} affects noise.

- In order to securely compute partial decryptions, each party adds error $r_j \leftarrow R_Q$ with coefficients sampled from $[\pm R]$ where $R = 2^\kappa(2\sigma PN^{1.5}n)$. Summing N parties' r_j samples scales the error by $\sqrt{N}$. So we get that the polynomial α_Q has noise bounded by

$$\sqrt{N}(2^\kappa(2\sigma PN^{1.5}n)) = 2^{\kappa+1}\sigma PN^2 n.$$

- Finally, to avoid wrap-around and thus ensure the correctness of rounding a polynomial in R_Q to a polynomial in R_P, we require a noise ceiling of

$$Q/P > 2^{\kappa+2}\sigma PN^2 n.$$

  and a modulus Q satisfying

$$Q > 2^{\kappa+2}\sigma P^2 N^2 n.$$

- Recall from the *Plaintext size* paragraph above, we know $P > 2^{\kappa+1}\tau^2 N$ so that it is large enough to encode integer values. With this, message size for P, we get

$$Q > 2^{\kappa+2}\sigma(2^{2\kappa+2}\tau^4 N^2)N^2 n = 2^{3\kappa+4}\tau^4\sigma N^4 n.$$

Alternatively, if the value of Q is mandated by security considerations, we get that the scheme can support the (parallel) execution of n multiplications with τ at most

$$\log\tau < \frac{\log Q - (3\kappa+4) - \log\sigma - 4\log N - \log n}{4}$$

many bits.

D Proof of Theorem 2

In this section, we complete the proof of security of our implementation of the triples functionality by reducing it to the semi-malicious security of the underlying threshold AHE scheme.

In Section 3, we described our simulator. Next, we argue indistinguishability of simulation. Assume for contradiction there exists an adversary $\mathcal{A}$, distinguisher D and polynomial $p(\cdot)$ such that, for infinitely many n, D distinguishes the following with probability at least $\frac{1}{p(n)}$.

- $\mathrm{IDEAL}_{\mathcal{F}_{\mathsf{ctriple}},\mathcal{S},\mathcal{Z}}(n)$
- $\mathrm{REAL}^{\mathcal{F}_{\mathrm{COM}},\mathcal{F}_{\mathrm{CP}}}_{\Pi_{\mathsf{ctriple}},\mathcal{A},\mathcal{Z}}(n)$

Fix an n for which this happens. We will arrive at a contradiction by constructing a (semi-malicious) adversary for the underlying threshold AHE scheme.

First, we observe that the real and the ideal world experiments proceed identically if some corrupted party P_j deviates anywhere in the protocol before the end of the triples generation phase. This is because the simulator follows the honest algorithm to simulate the honest parties and sets the triples based on the outputs of the algorithms, as in the real world. Furthermore, by the soundness of the $\mathcal{F}_{\text{CP}}$, we have that P_j will not succeed in making $\mathcal{F}_{\text{CP}}$ accept its proof. Therefore, conditioned on deviation, the real and ideal world are identically distributed.

Let dev denote the event the corrupted party deviates before the end of the triples generation phase. Now we consider modified experiments $\widetilde{\text{IDEAL}}$ and $\widetilde{\text{REAL}}$ that proceed identically to IDEAL and REAL, except that whenever the adversary deviates we set the output of the experiments to $\perp$. From the preceeding discussion, it follows that D distinguishes $\widetilde{\text{IDEAL}}$ and $\widetilde{\text{REAL}}$ with probability at least $\frac{1}{p(n)}$.

In the event all corrupted parties proceed honestly during the execution till the end of the triples generation phase, the real and ideal worlds are not the same - in the ideal world, the $\mathcal{F}_{\mathsf{ctriple}}$ functionality chooses the input for the honest parties after receiving the inputs of the corrupted parties. We observe that (conditioned on no party deviating), the distributions of the outputs of all the parties are identically distributed in the real and ideal world. Since the distinguisher distinguishes the real and ideal with probability $\frac{1}{p(n)}$, by an averaging argument, there exists a set of inputs for corrupted parties $\{(a_j, b_j, c_j)\}_{j \in U}$ such that the distinguisher distinguishes the real and ideal world with probability $\frac{1}{p(n)}$, conditioned on the adversary providing those inputs (recall that we have already conditioned on the adversary following the protocol honestly, which means that after the inputs and randomness are committed at the beginning, their inputs for the triples are determined). In slightly more detail, there is a partial transcript τ and randomness $r_{\mathcal{A}}$ for the adversary such that conditioned on τ, the distinguisher distinguishes with probability at least $\frac{1}{p(n)}$.

Using an averaging argument, we can further identify pairs of triples for honest parties $\{(\tilde{a}_j, \tilde{b}_j, \tilde{c}_j)\}_{j \notin U}$ and $\{(a_j, b_j, c_j)\}_{j \notin U}$ such that

- The adversary's provides $\{(a_j, b_j, c_j)\}_{j \in U}$ as its inputs and follows the protocol honestly, and
- $\sum_{j \notin U} c_j = \sum_{j \notin U} \tilde{c}_j$
- The adversary distinguishes the cases when the honest parties inputs are $\{(\tilde{a}_j, \tilde{b}_j, \tilde{c}_j)\}_{j \notin U}$ and $\{(a_j, b_j, c_j)\}_{j \notin U}$ with probability at least $\frac{1}{p(n)}$.

In order to construct an adversary for the TAHE scheme, we consider the machine B that incorporates D and non-uniform advice $(r_{\mathcal{A}}, \{(a_j, b_j, c_j)\}_{j \in U}, \{(a_j, b_j, c_j)\}_{j \notin U}, \{(\tilde{a}_j, \tilde{b}_j, \tilde{c}_j)\}_{j \notin U}$ and emulates an experiment of the ideal world. In the emulation, it chooses one honest party and forwards the messages in the key generation phase to the challenger of the TAHE scheme. It determines the challenge and semi-malicious queries using the information stored in the non-uniform advice. Finally, B outputs what the distinguisher D outputs. By construction, the advantage of B in the TAHE game is at least $\frac{1}{p(n)}$, and this contradicts the security of the TAHE scheme.