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.

2021 IEEE Symposium on Security and Privacy (SP) | 978-1-7281-8934-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/SP40001.2021.00025

***Abstract*—In this work, we design and implement the first protocol for distributed generation of an RSA modulus that can support thousands of parties and offers security against active corruption of an arbitrary number of parties. In a nutshell, we first design a highly optimized protocol for this scale that is secure against passive corruptions, and then amplify its security to withstand active corruptions using lightweight succinct zero-knowledge proofs. Our protocol achieves security with "identifiable abort," where a corrupted party is identified whenever the protocol aborts, and supports public verifiability.**

Our protocol against passive corruptions extends the recent work of Chen et al. (CRYPTO 2020) 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 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 minimize the work done by the parties, we employ a "publicly verifiable" coordinator that is connected to all parties and only performs computations on public data.

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 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 6 minutes and the active variant ran in under 25 minutes.

I. Introduction

We present a secure multiparty 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 has a basic variant that achieves security against either a *passive* (i.e., semi-honest) adversary, and an enhanced variant that achieves security against an *active* (i.e., malicious) adversary. In both cases, the adversary can corrupt all-but-one of the parties. We implemented both variants and benchmarked them with a thousand parties and more. As far as we know, this is the first implementation of secure multiparty computation at such a large scale; prior work reports at most 256 parties in a protocol involving a signature task [32].

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 distributed across a set of nodes, where corrupting up to a given threshold of nodes should not compromise security. The problem of generating a shared RSA modulus, introduced in the seminal work of Boneh and Franklin [16], [17], has recently regained attention owing to new and efficient constructions of verifiable delay functions (VDFs) [59], [15], [64], [56], [33] and zero-knowledge proof systems based on hidden-order groups [19].

Beyond the application to VDFs, distributed RSA key generation is a powerful primitive that is motivated by many applications in threshold cryptography; see [58], [31], [61], [27] for some earlier works in this area. One class of applications is generating keys for the public-key encryption scheme of Paillier [55], which is widely used in secure computation protocols. Paillier's encryption is a 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 [21]. Another application of sampling an RSA modulus arises in the context of generating a common reference string (CRS) for concurrent (i.e. UC-)secure computation [20]. This was demonstrated for general functions in [49] and for concrete functions such as the Fiat-Shamir authentication protocol [35], [34], set-intersection [48], and oblivious pseudorandom functions [48]. As noted above, more recent applications include setting up public parameters for VDFs. VDFs have many applications in decentralized systems, including leader election, generating public randomness beacons, proofs of replication, and many more.

© 2021, Megan Chen. Under license to IEEE.
DOI 10.1109/SP40001.2021.00025

A. Our Contribution

This paper pushes the boundaries of deployable secure computation protocols for useful tasks. We focus on RSA modulus generation because of its simplicity and relevance to a wide range of applications; however, the techniques we use can be applied more broadly. We were particularly motivated by the immediate application of obtaining a concrete VDF implementation that is useful in Ethereum's next generation consensus protocol. Here it is highly desirable to achieve $(n-1)$-security, namely security against any strict 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, it is desirable to accommodate at least one thousand participating parties in order to minimize the likelihood of corrupting all parties.

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 who 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, starting with the efficiency features.

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.medium` instances (4GB RAM, up to 2MBps uplink) for all parties and `r5dn.24xlarge` instances for the coordinator. For generating a 2048-bit modulus among 1,000 parties, the passive protocol executed in under 6 minutes and for 4,000 parties in 12 minutes (all measurements are averaged over multiple executions). For active security, the 1,000-party instance ran in roughly 24 minutes while the 4,000-party instance ran in 76 minutes. For the minimal 2-party setting, the protocol runs in 22 seconds in the passive case and 594 seconds in the active case.

Security challenges. An inherent limitation in the case of active security is that the adversary can always mount a "denial-of-completion" attack, even by corrupting only a single party [24]. 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 [47], [62], [8], [9] 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 [6], [7], [8], [9]. 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.

As we will further discuss in the related work section below (Section A), a natural bottleneck in large-scale secure computation is communication; most concretely efficient protocols require $O(|\mathrm{C}| \cdot n^2)$ communication, where $|\mathrm{C}|$ is the size of the circuit being computed and n is the number of parties, due to pairwise interaction for each gate. To circumvent this pairwise interaction barrier, we rely on threshold additively homomorphic encryption scheme that scales essentially linearly in the number of parties [37], [27]. Here we demonstrate that such protocols can achieve good concrete efficiency even in the case of active corruptions. The amortized per-party communication cost scales only logarithmically in the number of parties.

The coordinator model. In order to minimize the work performed by the n parties in the case of active corruptions, we introduce a new model where we employ a "publicly auditable" party, called a coordinator, who performs an *aggregate-and-broadcast* functionality in each round. This involves broadcasting the result of a simple computation on ciphertexts received from the n parties. What distinguishes the coordinator from all other parties is that it holds no secrets; its only role is to perform computations on public information. In principle, the coordinator could prepare an efficiently verifiable proof showing it performed its computation correctly. A simpler alternative that we pursue is to store the entire (signed) protocol transcript in a cloud service and allow anyone to validate it, as a substitute for such a work-saving proof.

In more detail, we first design a protocol that is secure against adversaries who actively corrupt the parties but only passively corrupt the coordinator. In order to have the actions of the coordinator publicly auditable, we apply a hash-and-sign paradigm where all parties and the coordinator hash their partial view at each round, sign it

and attach their signature to the next message. Intuitively, as long as one of the parties is honest, the coordinator cannot alter or reorder messages in the transcript since the honest party will detect this cheating.

An alternative viewpoint of our coordinator model is that we reduce the task of distributing the computation among the parties with active security to distributing the task of verification. By employing a coordinator who performs public operations, its actions that need to be verified are highly parallelizable. This makes it easy to distribute the verification task to a fleet of verifiers (which can be the parties themselves). Moreover, a cheating "verifier" cannot hurt security. In contrast, the goal of secure computation against an active adversary is far more challenging, since deviation from the protocol can affect both correctness and privacy.

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. In our implementation, we realize the authenticated channel via digital signatures and assume the coordinator and parties have in store the public keys of all parties they communicate with.

B. *Overview of Techniques*

The main bottleneck in scaling secure computation to a large number of parties is the cost of securely multiplying secret values. Recent works [50], [5], [43] 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 [63], [51] 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 [27] 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 [37], [27], 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 [30], are limited to covert security [29], or scale quadratically with the number of parties [50], [51], [60].

Our protocol design involves two stages: we start by 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 an (additive) share q_i 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-maliciously" secure, namely it is secure against an adversary who follows the protocol's instructions honestly except for 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 [22] based on the Chinese Remainder Theorem (CRT), which samples non-zero residues modulo small primes to en-

sure that the reconstruction is not divisible by these small primes. This approach increases the probability of hitting a valid prime to $\geq 1/60$. Upon computing the product, the parties complete the biprimality test as in the Boneh-Franklin test [16], [17].

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 $\mathcal{F}_{\mathrm{R-Triples}}$ [10]. 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" [41]. 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 [45], [38], [22].

Instantiating the building blocks. We instantiate our AHE scheme with a packed variant of an encryption scheme based on Ring Learning With Errors (RLWE) [52], [53]. 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 [22], 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 [13] and threshold fully homomorphic encryption from [18].

Our actively secure protocol relies on the Ligero zero-knowledge proof system [4], which has proof size square-root in the verification circuit size. While other proof systems can offer better asymptotic proof size [39], [42], [12], [11], [65], 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 [61] 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 IV).

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 $\mathcal{F}^{T}_{\mathsf{ctriple}}$. This functionality naturally extends the "certified OT" functionality from [46], [44] to the arithmetic setting and allows parties to obtain multiplication triples that are guaranteed to satisfy some global relation. (See Figure 1 for a more precise specification.) We present an efficient UC-secure implementation of $\mathcal{F}^{T}_{\mathsf{ctriple}}$ using any threshold AHE with security against semi-malicious adversaries, and then modularly analyze the security of our main protocol in the $\mathcal{F}^{T}_{\mathsf{ctriple}}$-hybrid model.

Relevant prior work. We briefly compare against two recent works in this area. Frederiksen et al. [38] 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. [22] 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 based only on OT (thus suffering quadratic communication complexity in n). Our protocol relies on their CRT techniques but improves by (a) developing a different, more efficient multiplier with linear communication complexity in n, (b) introduces the coordinator model, (c) achieves malicious security using a purely zero-knowledge technique, and (d) reports both a passive and actively secure implementations for up to 4000 parties, as opposed to [22] that does not report an implementation.

II. Preliminaries

A. Chinese Remainder Theorem (CRT)

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

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

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

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

Then f is an isomorphism from $\mathbb{Z}_N$ to $\mathbb{Z}_{p_1} \times \ldots \times \mathbb{Z}_{p_m}$ as well as an isomorphism from $\mathbb{Z}_N^*$ to $\mathbb{Z}_{p_1}^* \times \ldots \times \mathbb{Z}_{p_m}^*$ (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.

B. Threshold Homomorphic Encryption (THE)

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 1 (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* $(\mathrm{PK}_i, \mathrm{SK}_i)$ $\mathsf{Gen}(i)$ *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* $m_i = \mathsf{Dec}(\mathrm{SK}_i, c)$
- 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

[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.

3) The client C reconstructs the public key $\mathsf{PK} = \mathsf{Pub}(\mathsf{PK}_1, \ldots, \mathsf{PK}_n)$ from the public shares using Pub. In the case of a multiround key generation algorithm, the clients keep computing the values $\mathsf{PK}_i[r] = \mathsf{Gen}(\mathsf{SK}_i, [\mathsf{PK}[1], \ldots, \mathsf{PK}[r-1]])$ and sending them to the server which replies with $\mathsf{PK}[r] = \mathsf{Pub}(\mathsf{PK}_1[r], \ldots, \mathsf{PK}_n[r])$.
4) The client may encrypt message m using $\mathsf{Enc}(\mathsf{PK}, m) = \mathsf{Eval}(\mathsf{PK}, 0, f() = m, [])$
5) The client may homomorphically compute on messages using $\mathsf{Eval}(\mathsf{PK}, k, f, [c_1, \ldots, c_k])$ 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 $\mathsf{Dec}(i, \mathsf{SK}_i, c)$ 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. The security definition is presented in Appendix A.

III. Certified Triples Functionality

A core building block in our protocol is a functionality that generates multiplication triples (or Beaver triples [10]). 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 1).

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 an oblivious transfer (OT) protocol [46] 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 description is shown in Figure 1. Next, we realize our functionality via a threshold (additively) homomorphic encryption (THE) scheme. We give the security definition for THE in Section II-B. We discuss implementing these functionalities using Ring-LWE in Section V-A. The protocol can be found in Appendix D.

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

Functionality $\mathcal{F}^T_{\mathsf{ctriple}}$

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 $\mathcal{R}_{\mathrm{EXT}}$, an integer T, and domains $B_1, \ldots, B_T$. The functionality generates T triples where the ith 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 $\mathcal{S}$. Upon receiving a message $(\mathsf{generate}, sid, ssid, \mathcal{S}, \{\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j\}_{j\in U})$ from $\mathcal{S}$ 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 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 $(\mathsf{verify}, sid, ssid, P_j, \mathcal{R}_{\mathrm{EXT}}(x_j, \omega_j, \boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j))$ to all parties and $\mathcal{S}$. If an abort message was received on behalf of P_j, send $(\mathsf{abort}, sid, ssid, P_j)$ to C.

Fig. 1. The certified triples functionality.

IV. The Actively Secure Protocol

This section describes our actively-secure protocol for distributed RSA modulus generation.

At a high-level, our protocol has four phases: (1) presieve to sample primes, (2) compute biprimes, (3) sieve the easy non-biprimes and (4) sieve all non-biprimes. Most of these phases can be reduced to multiple invocations of secure multiplication. Towards this, we use the certified Beaver triples ideal functionality $\mathcal{F}_{\mathsf{ctriple}}$ (Figure 1), which provides the following guarantees: (1) sufficiently many Beaver multiplication triples are generated in parallel (2) at a later time, the parties must prove an arbitrary predicate over the sampled triples. With these

guarantees, we can modularly analyze our protocol in the $\mathcal{F}_{\mathsf{ctriple}}$-hybrid model. The triples generated in the first phase are consumed during the distributed sampling of the RSA modulus (presieving and candidate generation phases) and the GCD test.

To achieve active security, we take inspiration from the classic GMW paradigm [41], which provides a generic compiler from a passively-secure protocol to an actively-secure one via commitments and zero-knowledge proofs. The protocol $\Pi_{\mathrm{RSA-ML}}$ gives our actively-secure compilation from a passively-secure protocol (Section I-B). For GMW, parties commit to their input and randomness at the beginning of the computation, then zero-knowledge proofs are employed in each round of the protocol to enforce honest behavior in that round. We follow this paradigm with one important modification. Since parties do not provide inputs, they can simply commit to randomness at the beginning and provide ZK proofs at the end of the protocol, instead of in each round.

Our protocol begins with **Triples generation** via invoking $\mathcal{F}_{\mathsf{ctriple}}$, which in turn begins with a commitment to randomness. Then, the zero-knowledge proofs occur in **Certification and Σ-protocol** and involves invoking $\mathcal{F}_{\mathsf{ctriple}}$'s **Inputs for certification** and **Generating proofs** phases. The parties' NP statement for the zero-knowledge proofs is given in Section V-B. We will argue that this suffices to provide full security in our protocol.

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

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 biprime 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 biprime 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.

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

Notation. Let s be the statistical security parameter, $P_1, \ldots, P_n$ be the set of parties and C be a coordinator. Let T_1, T_2 be natural numbers such that the product $\prod_{k=1}^{T_1} d_k$ is $\ell - 2$ bits and the product $\prod_{k=1}^{T_2} d_k$ 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 by $\tau_1, \ldots, \tau_T$ the products of the primes in the corresponding buckets, i.e. $\prod_{i=1}^{T} \tau_i = \prod_{k=1}^{T_1} d_k$. 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 $\mathsf{receipt}$ message for $\mathcal{N}T + \lceil \ell/m \rceil \mathcal{N}' + \lceil (5\ell + 2\log n + s)/m \rceil J_{\mathrm{surv}}$ multiplication triples $[a], [b], [c]$, where J_{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 $\mathcal{N}' = \min_t |L_t|$.

Fig. 2. Actively secure generation of an RSA composite. (1/2)

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

CRT reconstruction. Each party P_j locally computes its shares of the prime candidates by invoking the CRT reconstruction algorithm. For the ith share of the primes, P_j first collects the ith 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 $(r^j_0, r^j_{i,1}, \ldots, r^j_{i,T})$ and $(\tilde{r}^j_0, \tilde{r}^j_{i,1}, \ldots, \tilde{r}^j_{i,T})$ w.r.t. the moduli $(4, \tau_1, \ldots, \tau_T)$ where P_1 sets $r^1_0 = \tilde{r}^1_0 = 3$ and the rest of the parties set $r^j_0 = \tilde{r}^j_0 = 0$.

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 oracle $\mathcal{F}_{\text{COIN}}$, the parties sample a random $\gamma_i \in \mathbb{Z}^*_{N_i}$ for each $i \in \mathcal{N}'$.
- 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 $Q_{\text{GCD}} > V \cdot N_i \cdot n$ such that it is a product of m-bit numbers, say $(B_1, \ldots, B_{\lceil \log Q_{\text{GCD}}/m \rceil})$. 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 $\alpha = \sum_j \alpha_j \bmod Q_{\text{GCD}}$ and sends α to the parties. Parties locally compute $z = \alpha \bmod N_i$. The parties eliminate N_i if $\text{GCD}(N_i, z) \neq 1$.

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 $\mathcal{F}_{\mathsf{ctriple}}$. Conditioned 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 [61].

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.

Fig. 3. Actively secure generation of an RSA composite. (2/2)

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 biprime (received from $\mathcal{F}_{\text{RSA-ML}}$) and produce a view identically distributed to the real world, or (2) the simulator embeds the biprime and when a deviation occurs, the simulator can complete the simulation with the knowledge of the factors of the target biprime. We remark that in the second case, the target biprime is not secure. We therefore extend the $\mathcal{F}_{\text{RSA-ML}}$ functionality to additionally accommodate a special request from the adversary upon which it will provide the factors of the biprime.

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).

V. Instantiating Our Primitives

A. Instantiating Our AHE Based on Ring-LWE

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

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 mth 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 $p_i - 1$ for all i. 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. We choose P and Q as follows. First, choose $Q = \Pi_{i=1}^h p_i$, where $p_1, \ldots, p_h$ is a set of h distinct primes. Next, we specify that P is the product of a subset of the primes in Q. That is, we choose a subset $i \subset [h]$, then set $P = \Pi_{i \subset [h]} p_i$.

For our implementation, we parameterize using values listed in Table I.

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 the polynomial $m(X) \in R_P$, i.e.

$\boldsymbol{m}_i = m(\zeta^{2i+1})$ for all $i \in [0, n-1]$. We compute $m(X)$ using the inverse of the discrete fast Fourier transform (FFT). Note that recovering $\boldsymbol{m}$ simply involves applying FFT to $m(X)$.

We observe that the FFT is a ring isomorphism that maps polynomial addition and multiplication (in R_P) to vector addition and pointwise multiplication 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 an 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 1, the protocol works as follows:

1) Secret key shares and randomness consists of triples $\mathsf{SK}_i = (a_i, s_i, e_i)$ 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, namely $\mathsf{Pub}(x_1, \ldots, x_n) = \sum_i \boldsymbol{x}_i \in R_Q$.
3) Two round key generation, 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 $\mathsf{PK} = (a, b) \in R_Q^2$. Once the public key is computed, the values a_i and e_i are no longer needed, and the secret key can be simply set to $\mathsf{SK}_i = s_i$.
4) The (randomized) distributed decryption algorithm $\mathsf{Dec}(\mathsf{SK}_i, (c, d); r)$, 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 uniformly 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:
$$\begin{aligned}&\mathsf{Eval}(\mathsf{PK}, 0, f()) = \mathsf{Enc}((a, b), x; u, v, w) \\ &\quad = \quad (a \cdot u + v, b \cdot u + w + (Q/P)x)\end{aligned}$$
where u, v, w χ.
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:
$$\mathsf{Eval}(\mathsf{PK}, n, f_\Sigma, [\boldsymbol{c}_1, \ldots, \boldsymbol{c}_n]) = \sum_i \boldsymbol{c}_i$$
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
$$\begin{aligned}&\mathsf{Eval}(\mathsf{PK}, 1, f_{y,z}, [\boldsymbol{c}]; u, v, w) \\ &\quad = \quad y\boldsymbol{c} + (au + v, bu + w + (Q/P) \cdot z)\end{aligned}$$
where u, v, w χ as in the encryption queries

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.

- 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. 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; the proofs can be found at [23].

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

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	$\lvert Q\rvert$	1302 bits
Plaintext Modulus Size	$\lvert P\rvert$	558 bits
Maximum number of bits for τ	$\mathsf{max_bits}(\tau)$	175 bits

TABLE I
RING-LWE CHOICE OF PARAMETERS.

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.*

Implementation parameters. Our code implements Ring-LWE operations using the open source library NFLlib [1], with parameters given in Table I.

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 $Q = 21$.

B. Implementing $\mathcal{F}_{\text{CP}}$ (Commit-and-Prove)

We will implement the commit-and-prove functionality using the Ligero zero-knowledge argument system [4] and Shoup's Σ-protocol [61]. 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 $a_i, s_i, e_i \in R_Q$ 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) The revealed a_i is consistent with the commitment.
   b) Each element of the vectors s_j and e_j is in the range $[-10\sigma, 10\sigma]$ except with very small probability. $10\sigma = 80$ for our parameters.
   c) 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 ith elements of $\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j$ from prime bucket B_i. We use a greedy strategy to identify $B_1, \ldots, B_n$ and the number of triples allocated for each category. 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 ith 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 $[-nB_i 2^\kappa, nB_i 2^\kappa]$.
   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 $m_1(\zeta_i^k) = \boldsymbol{a}_j[k] \bmod p_i$ for $i \in \{1, \ldots, 9\}$ and $1 \le k \le 65536$.
   f) Redefining notation so that $\sum_j \alpha_j = \alpha = (\alpha_1, \alpha_2)$, check that $\beta_j = \mathsf{Eval}(\mathrm{PK}, 1, f_{lin}(\boldsymbol{b}_j, \boldsymbol{c}'_j, \cdot), \alpha) = (m_2 \cdot \alpha_1 + a \cdot u_2 + v_2, m_2 \cdot \alpha_2 + b \cdot u_2 + w_2 + (Q/P)m_3)$, where $m_2(\zeta_i^k) = \boldsymbol{b}_j[k]$, $m_3(\zeta_i^k) = \boldsymbol{z}_j[k] - \boldsymbol{c}_j[k] \bmod p_i$ for $i \in \{1, \ldots, 9\}$ and $1 \le k \le 65536$.
   g) Redefining notation so that $\sum_j \beta_j = \beta = (\beta_1, \beta_2)$, check that $d_j = \mathsf{Dec}(j, \mathrm{SK}_j, \beta) = \delta_{j,1} \cdot \beta_2 - \beta_1 \cdot s_j + r$, 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 $r_{i,t}^j, \tilde{r}_{i,t}^j$ such that the triples were used correctly in the multiplication protocol using triples to compute $(\sum_i r_{i,t}^j) \times (\sum_i \tilde{r}_{i,t}^j) \bmod B_t$ and that $0 < r_{i,t}^j, \tilde{r}_{i,t}^j < B_t$.
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 the prime shares $p_{i,j}, q_{i,j}$ are 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 [61].
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.

Proving consistency between the three proof systems. We incorporated our NP statement using three proof systems that have overlapping 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. We identify the overlapping parts of the witnesses.

(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. On the other hand, the main proof factors each of the equations into the 21 moduli by taking all equations/variables modulo the corresponding prime. Hence, the first modulus 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 incorporate 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 $[p_1]$, which we incorporate in the witness for the first modulus 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 of 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 $[p_1]$. After computing the output of this linear function modulo p_1, we compute the output modulo $p_2 \cdots p_{21}$, then compare the values. To compute this function over $p_2 \cdots p_{21}$, we include a large multiple of p_1 to the inner product so that only the value mod p_1 is revealed. Finally, we remark 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 to $\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j$ at the beginning.

VI. 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. Table II lists the external libraries required. Our implementation's networking layer defines primitives for sending messages and awaiting replies and a separate encryption protocol performs the computation and tests the data. A substantial effort was made to properly select Ring-LWE, commitment, ZK and hashing parameters in order to achieve roughly 128-bits of security.

Purpose	Purpose
Networking	ZeroMQ [2]
Ring-LWE data structures	NFLlib [1]
Serialization	Boost

TABLE II
External code libraries

Software engineering. We use ZeroMQ [2], a fast request-reply concurrency framework for all networking operations. 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.

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.

On the coordinator side, we set the following time bounds. The coordinator waits a `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 `cleanup_time` = 5secs, 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).

A. Experiments

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, 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

Parties	Passive ($\mu \pm \sigma$ s)	Active ($\mu \pm \sigma$ s)	Registration (s)	# Runs (passive/active)
2	20.5 ± 0.9	594.3 ± 1.1	0.3	20 / 10
5	52.4 ± 3.7	785.9 ± 5.5	0.8	20 / 10
10	53.3 ± 1.9	788.5 ± 3.3	0.8	20 / 10
20	56.6 ± 2.3	797.7 ± 6.6	0.8	20 / 11
50	67.9 ± 6.6	808.8 ± 8.6	1.0	20 / 16
100	91.4 ± 5.3	832.3 ± 5.5	3.9	20 / 9
200	133.5 ± 12.2	884.4 ± 14.2	1.0	15 / 9
500	219.8 ± 5.9	970.0 ± 6.1	0.9	9 / 6
700	279.7 ± 4.9	1069.8 ± 9.8	61.4	5 / 5
1000	352.0 ± 14.0	1429.2 ± 0.0	1.6	3 / 1
2000	817.8 ± 0.0	2966.8 ± 0.0	2.0	1 / 1
4046	684.2 ± 0.0	4580.7 ± 0.0	158.7	1 / 1

TABLE III
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.

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	175,720
$AXBY$ Value	1,597,389
Modulus Candidate	1,908,015
$AXBY$ Value	1,597,389
All others	1396
Total Semihonest	71,340,451
Gather Public Data	143,112,726
Gather Proofs (21)	229,568,412
Total Active	444,021,589

TABLE IV
PER PARTY COMMUNICATION COMPLEXITY OF THE PROTOCOL

`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.

2) Empirical Results: Table III, 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[2].

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 (uplink speed $\geq$ 1 Mbps) 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 IV 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 III, the active security portion of the protocol dominates the running time. In this section, we further analyze the components of this step.

[2]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.

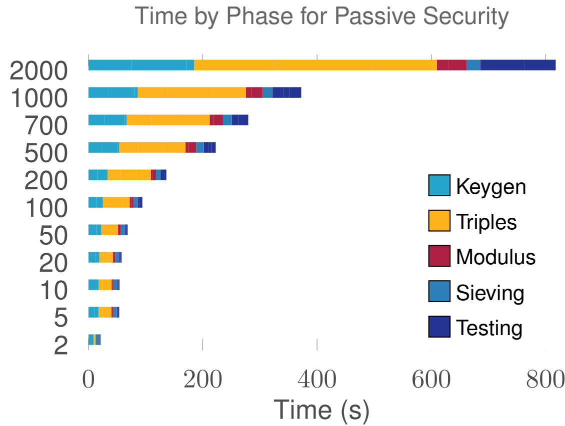


Fig. 4. Cumulative timing per step of the protocol as the number of parties increases

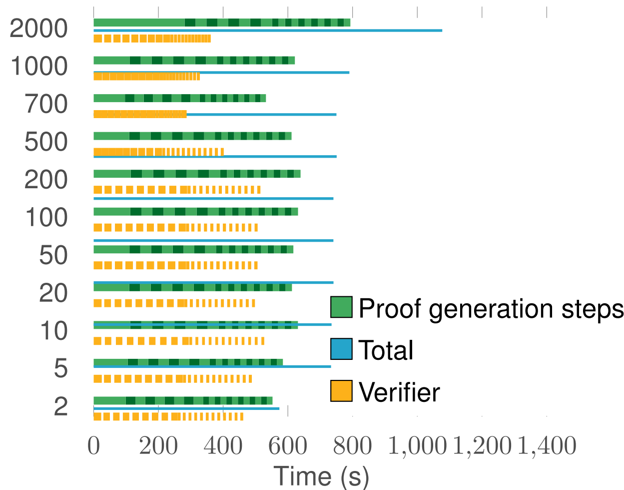


Fig. 5. Cumulative timing per step of the active protocol as the number of parties increases.

Figure 5 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, 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.

VII. 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. We thank the anonymous reviewers of IEEE S&P for insightful comments.

Carmit Hazay is supported by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office, and by ISF grant No. 1316/18. Yuval Ishai is supported by ERC Project NTSC (742754), NSF-BSF grant 2015782, BSF grant 2018393, and a joint Israel-India grant. Daniele Micciancio is supported in part by NSF award 1936703. abhi shelat is supported by NSF grant TWC-1646671.

References

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

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

[3] 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.

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

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

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

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

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

[9] Carsten Baum, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. Efficient constant-round MPC with identifiable abort and public verifiability.

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

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

[12] 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.

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

[14] 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.

[15] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *CRYPTO*, pages 757–788, 2018.
[16] Dan Boneh and Matthew K. Franklin. Efficient generation of shared RSA keys (extended abstract). In *CRYPTO*, pages 425–439, 1997.
[17] Dan Boneh and Matthew K. Franklin. Efficient generation of shared RSA keys. *J. ACM*, 48(4):702–722, 2001.
[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.
[19] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent snarks from DARK compilers. In *EUROCRYPT*, pages 677–706, 2020.
[20] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.
[21] 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.
[22] Megan Chen, Ran Cohen, Jack Doerner, Yashvanth Kondi, Eysa Lee, Schuyler Rosefield, and abhi shelat. Multiparty generation of an RSA modulus. *Manuscript*, 2020.
[23] Megan Chen, Carmit Hazay, Yuval Ishai, Yuriy Kashnikov, Daniele Micciancio, Tarik Riviere, Abhi Shelat, Muthuramakrishnan Venkitasubramaniam, and Ruihan Wang. Diogenes: Lightweight scalable RSA modulus generation with a dishonest majority. *IACR Cryptol. ePrint Arch.*, 2020:374, 2020.
[24] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *STOC*, pages 364–369, 1986.
[25] 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.
[26] Don Coppersmith. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. *J. Cryptology*, 10(4):233–260, 1997.
[27] R. Cramer, I. Damgård, and J. B. Nielsen. Multiparty computation from threshold homomorphic encryption. In *EUROCRYPT*, pages 280–299, 2001.
[28] I. Damgård and G. L. Mikkelsen. Efficient, robust and constant-round distributed RSA key generation. In *TCC*, pages 183–200, 2010.
[29] 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.
[30] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, pages 643–662, 2012.
[31] Yvo Desmedt. Threshold cryptography. *European Transactions on Telecommunications*, 5(4):449–458, 1994.
[32] 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.
[33] Naomi Ephraim, Cody Freitag, Ilan Komargodski, and Rafael Pass. Continuous verifiable delay functions. *IACR Cryptology ePrint Archive*, 2019:619, 2019.
[34] U. Feige, A. Fiat, and A. Shamir. Zero-knowledge proofs of identity. *J. Cryptology*, 1(2):77–94, 1988.
[35] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, pages 186–194, 1986.
[36] Yair Frankel, Philip D. MacKenzie, and Moti Yung. Robust efficient distributed rsa-key generation. In *STOC*, pages 663–672, 1998.
[37] Matthew K. Franklin and Stuart Haber. Joint encryption and message-efficient secure computation. *J. Cryptology*, 9(4):217–232, 1996.
[38] 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.
[39] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *EUROCRYPT*, pages 626–645, 2013.
[40] Niv Gilboa. Two party RSA key generation. In *CRYPTO*, pages 116–129, 1999.
[41] 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.
[42] Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT*, pages 305–326, 2016.
[43] Carmit Hazay, Yuval Ishai, Antonio Marcedone, and Muthuramakrishnan Venkitasubramaniam. LevioSA: Lightweight secure arithmetic computation. In *To appear CCS*, 2019.
[44] 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.
[45] 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.
[46] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, Manoj Prabhakaran, and Amit Sahai. Efficient non-interactive secure computation. In *EUROCRYPT*, pages 406–425, 2011.
[47] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multi-party computation with identifiable abort. In *CRYPTO*, pages 369–386, 2014.
[48] 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.
[49] S. Jarecki and V. Shmatikov. Efficient two-party secure computation on committed inputs. In *EUROCRYPT*, pages 97–114, 2007.
[50] 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.
[51] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT*, pages 158–189, 2018.
[52] 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.
[53] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. A toolkit for ring-LWE cryptography. In *EUROCRYPT*, pages 35–54, 2013.
[54] Takashi Nishide and Kouichi Sakurai. Distributed paillier cryptosystem without trusted dealer. In *WISA*, pages 44–60, 2010.
[55] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, pages 223–238, 1999.
[56] Krzysztof Pietrzak. Simple verifiable delay functions. In *ITCS*, pages 60:1–60:15, 2019.
[57] Guillaume Poupard and Jacques Stern. Generation of shared RSA keys by two parties. In *ASIACRYPT*, pages 11–24, 1998.
[58] Tal Rabin. A simplified approach to threshold and proactive RSA. In *CRYPTO*, pages 89–104, 1998.
[59] Ronald R. Rivest, Adi Shamir, and David A. Wagner. Time-lock puzzles and timed-release crypto. 1996.
[60] Dragos Rotaru, Nigel P. Smart, Titouan Tanguy, Frederik Vercauteren, and Tim Wood. Actively secure setup for SPDZ. *IACR Cryptology ePrint Archive*, 2019:1300, 2019.
[61] Victor Shoup. Practical threshold signatures. In *EUROCRYPT*, pages 207–220, 2000.
[62] Gabriele Spini and Serge Fehr. Cheater detection in SPDZ multiparty computation. In *ICITS*, pages 151–176, 2016.
[63] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *CCS*, pages 39–56, 2017.

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

[65] 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.

APPENDIX

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 the 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 2. *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*[3] *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.*[4]
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. Semi-malicious 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*

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

   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.

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.

A. Related Work

We recount the main prior works about distributed RSA key generation. The seminal work by Boneh and Franklin [16], [17] initiated the line of research on concretely efficient protocols for this task. They introduced the first non-trivial technique for choosing an

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

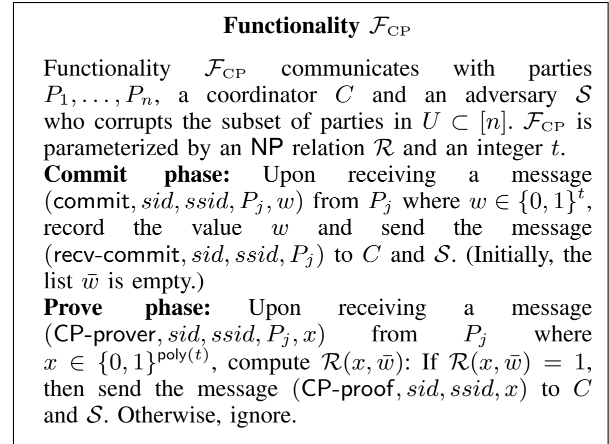
[4] Specifying all queries at the outset is what makes the definition *non-adaptive*. In a fully adaptive definition the adversary can choose each query after receiving the answer to previous queries. The non-adaptive security definition is enough for our purposes as the protocol is ultimately made secure 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.

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 adversaries. Two followup papers [36], [54], still in the honest-majority setting, strengthened this result and obtained security against *active* adversaries. Additional solutions for testing primality in the multi-party setting appear in [3], [28]. Unlike previous approaches that relied on the biprimality testing procedure from [17], 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 [25] 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 [26], [14]. The problem was solved by Gilboa [40] who presented a protocol in the passive two-party setting, adapting the technique from [17]. 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. [14] presented the first protocol for the active two-party setting, but did not provide a proof of security. Concurrently, Poupard and Stern [57] 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. [45]. 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. [38] 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. [22] further improves this protocol and extends it to more than two parties; however, no concrete analysis is provided. Moreover, the communication complexity of this protocol scales quadratically with the number of parties.

B. Commit-and-Prove Functionality



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

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

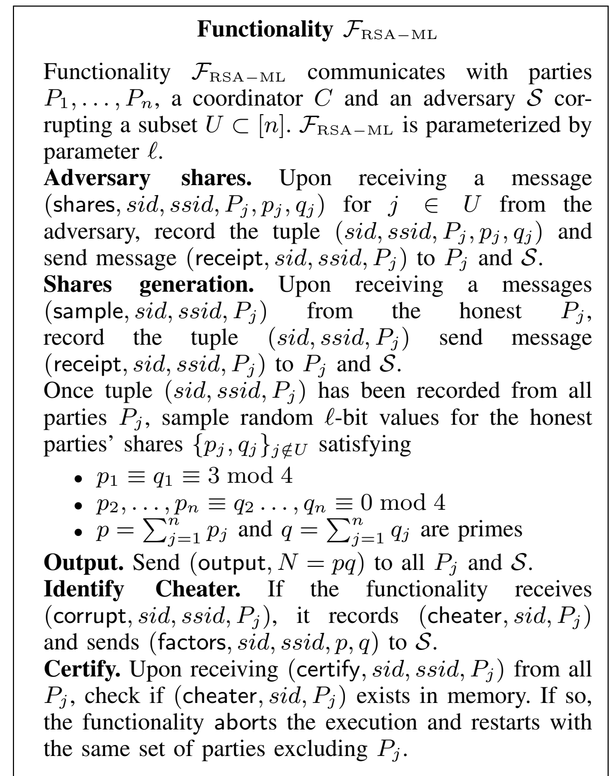
Commit phase: Upon receiving a message $(\mathsf{commit}, sid, ssid, P_j, w)$ from P_j where $w \in \{0,1\}^t$, record the value w and send the message $(\mathsf{recv\text{-}commit}, sid, ssid, P_j)$ to C and $\mathcal{S}$. (Initially, the list $\bar{w}$ is empty.)

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.

Fig. 6. Functionality for commit-and-prove.

C. The RSA Composite Functionality

Functionality $\mathcal{F}_{\mathrm{RSA-ML}}$, Figure 7, 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.



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

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

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 $\mathcal{S}$.

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

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

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 $\mathcal{S}$.

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.

Fig. 7. The shared RSA functionality in the active setting.

D. Protocol for Certified Triples

PROTOCOL $\Pi^{T}_{\mathsf{ctriple}}$ CERTIFIED TRIPLES GENERATION

Notations. The protocol communicates with parties $P_1, \ldots, P_n$ and a coordinator C, and is parameterized by modulus M, and relies on a threshold additively-homomorphic public key encryption scheme $(\mathsf{Gen}, \mathsf{Pub}, \mathsf{Eval}, \mathsf{Dec}, \mathsf{Rec})$ over the plaintext ring $\mathbb{R}$ with packing factor T with an r_{Enc}-round key-generation protocol. Let $B_1, \ldots, B_T$ be the corresponding domains from which each of the T vectors needs to be 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 $\mathcal{F}_{\mathrm{CP}}$. $\mathcal{F}_{\mathrm{CP}}$ sends $(\mathsf{receipt}, sid, ssid)$ to C.

AHE 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. C broadcasts $\mathrm{PK}[1] = \mathsf{Pub}(\mathrm{PK}_1[1], \ldots, \mathrm{PK}_n[1])$ 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. C broadcasts $\mathrm{PK}[r] = \mathsf{Pub}(\mathrm{PK}_1[r], \ldots, \mathrm{PK}_n[r])$ 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 $\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j \in \mathbb{Z}_{B_1} \times \cdots \times \mathbb{Z}_{B_T}$. In addition, they sample 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) Once C receives input from all parties, it 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 sends $\beta_j = \mathsf{Eval}(\mathrm{PK}, 1, f_{lin}(\boldsymbol{b}_j, \boldsymbol{c}'_j, \cdot), \alpha)$ and sends β_j to C where $\boldsymbol{c}'_j = \boldsymbol{z}_j - \boldsymbol{c}_j$. f_{lin} is defined over three inputs $\boldsymbol{b}_j, \boldsymbol{c}'_j$ and a plaintext vector $\boldsymbol{m}$ (that corresponds to decrypted ciphertexts α), and takes the ith component of $\boldsymbol{b}_j, \boldsymbol{c}_j$ and $\boldsymbol{m}$, say b, c, m and returns as the ith component of the output vector as $bm + c$.
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 ith element of $\boldsymbol{c}''_1$ is set to $c^i_1 + w_i \bmod \mathbb{Z}_{B_i}$. The rest of the 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 the transcript of interaction of P_j 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 $(\omega_j, R_j, \boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j)$ and outputs 1 if $(x_j, (\omega_j, \boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j)) \in \mathcal{R}_{\mathrm{EXT}}$, $(\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{c}_j)$ is consistent with R_j and τ_j is consistent with the honest party's code using randomness R_j). $\mathcal{F}_{\mathrm{CP}}$ delivers the result to C. If for any party P_j the C does not receive a confirmation from $\mathcal{F}_{\mathrm{CP}}$, it identifies P_j as a faulty party and broadcasts this to all parties.

E. Proof Sketch of Theorem 2

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 $\mathcal{F}_{\mathrm{CP}}$.

– Simulating the AHE setup phase: The simulator $\mathcal{S}$ simulates the messages from uncorrupted parties honestly. If 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 $\mathcal{F}_{\mathsf{ctriple}}$.

If 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 is completes the emulation till 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 w_j from the commit command. Then, for every P_j $(j \in U)$ 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 $(\mathsf{certify}, sid, P_j, (x_j, w_j))$ to $\mathcal{F}_{\mathsf{ctriple}}$. Otherwise, it sends (abort, P_j). $\square$

F. Proof of Theorem 5

Consider a security game as described in Definition 2 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

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

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.

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

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

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

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

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 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.