High-Throughput Universally Composable Threshold FHE Decryption

Guy Zyskind
Fhenix
Tel Aviv, Israel
guy@fhenix.io

Doron Zarchy
Fhenix
Tel Aviv, Israel
doron@fhenix.io

Max Leibovich
Fhenix
Tel Aviv, Israel
max@fhenix.io

Chris Peikert
Fhenix
University of Michigan
Ann Arbor, MI, USA
chris@fhenix.io

ABSTRACT

Threshold Fully Homomorphic Encryption (FHE) enables arbitrary computation on encrypted data, while distributing the decryption capability across multiple parties. A primary application of interest is low-communication multi-party computation (MPC), which benefits from a fast and secure threshold FHE decryption protocol.

Several works have addressed this problem, but all existing solutions rely on "noise flooding" for security. This incurs significant overhead and necessitates large parameters in practice, making it unsuitable for many real-world deployments. Some constructions have somewhat better efficiency, but at the cost of weaker, non-simulation-based security definitions, which limits their usability and composability.

In this work, we propose a novel threshold FHE decryption protocol that avoids "noise flooding" altogether, and provides strong simulation-based security. Rather than masking the underlying ciphertext noise, our technique securely removes it through an efficient MPC rounding procedure. The cost of this MPC is mitigated by an offline/online design that preprocesses special gates for secure comparisons in the offline phase, and has low communication and computation in the online phase. This approach is of independent interest, and should also benefit other MPC protocols (e.g., secure machine learning) that make heavy use of non-linear comparison operations.

We prove our protocol secure in the Universal Composability (UC) framework, and it can be generally instantiated for a variety of adversary models (e.g., security-with-abort against a dishonest majority, guaranteed output delivery with honest majority). Compared to the state of the art, our protocol offers significant gains both in the adversary model (i.e., dishonest vs. honest majority) and practical performance: empirically, our online phase obtains approximately 20,000× higher throughput, and up to 37× improvement in latency.

1 INTRODUCTION

Since the introduction of the first Fully Homomorphic Encryption (FHE) scheme [29] about 16 years ago, FHE has received intense sustained interest from both academia and industry. This is of no surprise, since FHE enables a wide array of powerful secure applications. In its simplest form, FHE enables a client to delegate arbitrary computations on its private data to an untrusted server, without revealing anything about the data. More specifically: the client first *encrypts* its data using an FHE scheme, yielding a ciphertext (which hides the data) that is sent to the server. The server then *homomorphically evaluates* any desired function on the ciphertext, yielding a new ciphertext that encrypts the value of the function on the data, which is returned to the client. Finally, the client uses its secret key to *decrypt* the derived ciphertext, thus recovering the function value.

Another very important application of FHE is to *secure multi-party computation* (MPC), which enables several parties to compute a function on their individual secret inputs, without revealing anything but the final result. The standard approach here relies on *threshold* FHE, which securely distributes the decryption key across all the parties (typically, using some form of secret sharing). Each party encrypts its own input, then the desired function is homomorphically evaluated on these ciphertexts, then a qualified subset of the parties jointly runs a *threshold decryption* protocol (using their shares of the secret key) to recover the final answer. This threshold decryption is the most delicate step, since it must reveal the underlying plaintext and nothing more (about the secret key or the like).

Every well studied FHE scheme is based on some variant of the Learning With Errors (LWE) problem [41]. LWE is believed to resist quantum attacks, and is the foundation for countless other cryptographic constructions, from basic public-key encryption to countless powerful and versatile concepts (see [40] for a survey). In general, most of these constructions can benefit from being "thresholdized," i.e., distributing their secret keys and decryption operations across several entities, to eliminate single points of failure.

Given the importance of threshold decryption for FHE and LWE-based constructions more broadly, many works have addressed this problem in recent years (e.g., [2, 4, 5, 20, 33, 35], to name just a few). However, all of these protocols are either too inefficient for most applications [2, 4, 20, 35]; rely on new assumptions or security models [33, 39, 47]; or settle for less than simulation security, instead resorting to weaker security definitions (e.g., [7, 18, 21]) that can be insufficient for the desired applications [39].

Noise flooding and its discontent. While all these works differ in some respects, every one that applies to FHE uses some form of *noise flooding*. The basic reason for this arises from the structure of "standalone" FHE/LWE decryption, which works in two steps:

(1) First, we compute the inner product $\langle \mathbf{c}, \mathbf{s} \rangle$ of the ciphertext $\mathbf{c}$ and secret-key vector $\mathbf{s}$. This yields a "noisy encoding" of the plaintext μ, of the form $\mu + e$ for some small enough error (or noise) term e.
(2) Then, we "round away" the noise e to recover the plaintext μ itself.

In other words, the overall decryption function is the *rounded inner product* of the ciphertext and the secret key.

In the threshold setting, each decrypting party typically holds a share of the secret key, under some linear secret-sharing scheme. Thanks to the linearity, the parties can locally compute a sharing of the inner product $\langle \mathbf{s}, \mathbf{c} \rangle = \mu + e$, i.e., the noisy plaintext. At this point, the parties could simply reveal their shares to obtain the noisy plaintext, and round away the error. Alas, this would be *fatal* for security, since the noise (combined with some knowledge of the encryption randomness) leaks secret information that can even be used to recover the secret key. In particular, there is no hope of simulating this threshold decryption protocol using knowledge of the plaintext alone.

Noise flooding addresses this issue by having the decrypting parties add a large amount of extra noise before revealing their decryption shares. This statistically masks, or "floods," the original noise e, making a simulation-based security proof possible. However, this comes at a high cost: it requires the original noise rate to be very small—asymptotically negligible—and to compensate for this, the other LWE parameters must be made much larger to ensure adequate security. This results in significantly larger ciphertexts and slower homomorphic operations, compared to the standalone setting.

Indeed, the current state-of-the-art proposed by Dahl *et al.* [20], which is the only scheme having an adequate form of (simulation-based) security and reasonable performance in practice, walks this fine line by using *bootstrapping* to switch to small noise rate and large LWE parameters *just prior to* threshold decryption (and not for regular FHE ciphertexts and operations). Yet this still imposes a high computational cost: a single decryption takes hundreds of milliseconds on a well-equipped 128-core machine. More importantly, being compute-bounded like this limits the *throughput* of the protocol, namely, the number of possible decryptions per server per unit of time—regardless of the speed of the network.

Given the growing interest in practical threshold FHE systems (e.g., [36, 44, 46, 49]), there is a clear need for approaches that are more efficient and scalable. In particular, a *low-computation* and *low-communication* protocol would have much higher throughput than existing ones, even if it required slightly more rounds of interaction. In fact, if the latency of the extra rounds was less than the savings in computation time, such a protocol could even have have better latency as well. This is the approach we pursue in this work.

1.1 Our Contribution

We propose a novel approach to threshold decryption for widely used FHE schemes, and LWE-based encryption more generally. Our protocol uses small LWE parameters—matching those for the standalone setting, in fact—and requires low computation and communication, at the cost of a slight increase in the rounds of interaction. As shown by our evaluation, it thereby achieves substantially higher throughput than the state of the art, and, in most realistic network settings, even lower latency as well. Crucially, we accomplish all this without compromising on security, obtaining strong simulation-based security in the universal composability (UC) framework [10] for a variety of adversary models.

In contrast to all prior work on threshold FHE, our protocol completely avoids the costly "noise flooding" technique—in fact, it adds no additional noise at all (thus preserving the LWE parameters of standalone schemes). Instead, we devise a novel *MPC-based approach* for removing the ciphertext error during decryption. High efficiency is obtained from an offline-online structure, where the offline phase can be executed at any time before the ciphertext to be decrypted is available, and the online phase takes a small number of rounds and uses very little computation and communication. We also highly optimize the offline phase and show empirically that it is efficient enough to support real-time usage in practice.

At a more technical level, some highlights of our results include the following.

(1) Our protocol is designed in the abstract Arithmetic Black Box (ABB) model (see, e.g., [19, 25]), making it simple to express and analyze, and very generally instantiable for a variety of adversary models (e.g., dishonest majority, honest majority, semi-honest or robust, static or adaptive corruptions, etc.).
(2) Empirically, our evaluation shows that our protocol is orders of magnitude more efficient than the state of the art [20]. In an instantiation of our protocol with the SPD$\mathbb{Z}_{2^k}$ protocol for dishonest majority [19], we achieve up to 20,000× higher throughput, and up to 37× lower latency for decrypting a single ciphertext.
(3) As a main ingredient, we develop new efficient MPC protocols for comparison-type functions (less-than, mod, etc.) for moderate-size inputs (e.g., 64–256 bits). These protocols require very low communication and round complexity in their online phase, at the cost of a moderate but very practical offline phase. This contribution shouuld be of independent interest for many other MPC protocols and applications, such as secure Machine Learning (e.g., [34, 43, 48]).

1.2 Technical Overview

The basic approach behind our protocol is to entirely avoid noise flooding, and instead use MPC to *completely remove* the noise before anything is revealed to the parties. This sidesteps the complexities of managing noise, and dramatically simplifies the security analysis—instead of subtle arguments about noise distributions, we just prove that our MPC protocol is UC-secure. Therefore, the parties effectively have access only to an idealized decryption function, and in particular they learn nothing except the decrypted plaintext. We do not need to resort to novel security models or special game-based definitions.

Of course, securely removing noise requires secure comparisons, which typically involves bit-level operations that can be expensive if implemented naïvely in MPC [12, 22, 23, 38]. To avoid such a cost, we introduce a novel approach that trades a somewhat larger offline phase for a much cheaper online phase. More concretely, in the offline phase we build special preprocessed (i.e., input-independent) "gates" of moderate size, so that in the online phase the parties can securely perform the rounding step using just two "layers" of these gates. Our empirical evaluation shows that this protocol achieves much higher throughput and even significantly lower latency than prior noise-flooding approaches, delivering fast threshold FHE decryption in practice.

We next give a high-level technical overview of our threshold decryption protocol. For simplicity of presentation and generality of instantiation, it operates in the abstract Arithmetic Black-Box (ABB) model for the ring $\mathbb{Z}_{2^k}$. Essentially, the ABB provides an idealized trusted party that: stores secret values in $\mathbb{Z}_{2^k}$, including random bits that it generates; computes desired products and linear combinations of stored values; and reveals values upon request by the honest parties. We stress that powers of two are popular choices of moduli in FHE and LWE-based cryptography, but we can also trivially support any other choice as well, by cheaply "modulus-switching" to a power of two just before decryption.

Let $q = 2^k$ be the ciphertext modulus and $p = 2^m \ll q$ be the plaintext modulus, and let $L = q/p$. The secret key is a vector $\mathbf{s}$ over $\mathbb{Z}_q = \mathbb{Z}_{2^k}$, which is stored in the ABB. When the honest parties wish to decrypt a ciphertext $\mathbf{c}$, they proceed as follows:

(1) **Inner product.** They instruct the ABB to internally compute the "noisy message" $z = \langle \mathbf{c}, \mathbf{s} \rangle = L \cdot \mu + e$, where $\mu \in \mathbb{Z}_p$ is the message and $e \in [0, L)$ is the noise. This is done by taking a linear combination, which is purely local computation in a typical instantiation of the ABB.
(2) **Error extraction.** They engage in a lightweight MPC protocol, in which the ABB computes and stores the error, as $e = z \bmod L$ (still stored as an element of $\mathbb{Z}_{2^k}$).
(3) **Message recovery.** They instruct the ABB to cancel out the noise value by computing $z - e = L \cdot \mu$ (again using linearity), then have ABB open this value, which reveals μ itself.

The only nontrivial step is the secure computation (by ABB) of $z \bmod L$ from z. Doing so naïvely requires concretely too many rounds and/or too much communication for our purposes [22]. Instead, we take a novel approach of computing the mod operation using just two layers of moderate-size preprocessed "gates," for very specially designed functions; see Section 3.3 for details.

An interesting side note is that these gates are inspired by works on Function Secret Sharing (FSS), which allows to perform MPC with very little or even no interaction [8, 9, 51]—but usually at high computational and/or storage cost, and often requiring novel hardness assumptions. While precomputing our gates is somewhat (but not very) costly, it allows us to compute secure comparisons using a small constant number of rounds and communication in the online phase. For example, for $k = 64$, which is a value of interest in this work and in many others (given the prevalence of 64-bit architectures), the online phase requires only three sequential openings from the ABB. Finally, since the online phase has very low communication and computation, it allows us to scale up the number of parallel decryptions to obtain high throughput.

2 PRELIMINARIES

2.1 Notation

For a positive integer n, define $[n] = \{0, 1, \ldots, n-1\}$. (We caution that this notation almost overlaps with the one for stored values in $\mathcal{F}_{\text{ABB}}$, but the intended meaning should always be clear from context.) Let $\mathbb{Z}_A$ be the quotient ring of integers modulo a positive integer A; in this work, A is always a power of two.

2.2 Arithmetic Black Box (ABB) Model

In the MPC literature, the Arithmetic Black Box (ABB) model is a widely used abstraction for secure low-level arithmetic operations [19, 24, 27, 32]. The ABB supports basic operations, including addition and multiplication, on elements of some specific ring or field; in this work, we exclusively use rings of the form $\mathbb{Z}_{2^k}$. The ABB greatly simplifies the design, presentation, and modularity of protocols that use it, by abstracting away the underlying cryptographic mechanisms like secret-sharing schemes and MACs.

The functionality $\mathcal{F}_{\text{ABB}}$ (Figure 1) formalizes the Arithmetic Black Box model as an idealized trusted party in the UC framework, defining an abstraction for secure, reactive arithmetic computations. It allows parties to input, perform arithmetic operations on, and reveal secret values that are stored within the functionality. We present just $\mathcal{F}_{\text{ABB}}$'s "core" definition, which can be paired with any standard "shell" corresponding to the adversary model, e.g., honest majority with fairness, dishonest majority with abort, adaptive security, etc.

Formally, we present our decryption protocol in the $\mathcal{F}_{\text{ABB}}$-hybrid model using the UC framework, and show that it perfectly realizes an abstract decryption functionality having the same shell as $\mathcal{F}_{\text{ABB}}$. This makes our protocol highly general and applicable, since it automatically inherits the specific efficiency and security features of any particular realization of $\mathcal{F}_{\text{ABB}}$ with a given shell.

For convenience of usage, we have slightly modified the interface of $\mathcal{F}_{\text{ABB}}$ as follows:

- We have elided the Input command, because our protocols make no use of it.
- We have added the RandBit command, which generates and stores a uniformly random bit with some specified identifier. This has known implementations [22, 27, 42] that are compatible with existing realizations of $\mathcal{F}_{\text{ABB}}$ for $\mathbb{Z}_{2^k}$.
- We have made the Open command a "truncated opening," where only the *least-significant* l bits of the stored value are revealed, for some desired $l \le k$. This enhancement can be implemented generically (in terms of the standard full opening) by using RandBit and linear operations to randomize the most-significant $k - l$ bits of the revealed value. Or, in some realizations of $\mathcal{F}_{\text{ABB}}$ this can be implemented more directly and efficiently; e.g., with Shamir sharing over Galois rings, revealing just the low l bits of each share opens just the low l bits of the shared value.

Notation. When defining protocols that use $\mathcal{F}_{\text{ABB}}$, it is cumbersome to use its formal interface, so for convenience we instead use some more natural notation.

We write $[x]_k$ to denote a value $x \in \mathbb{Z}_{2^k}$ that has been stored in the functionality $\mathcal{F}_{\text{ABB}}$; this value is implicitly identified by some

Functionality $\mathcal{F}_{\text{ABB}}$ for ring $R = \mathbb{Z}_{2^k}$

- **Random Bit Generation:** On input $(\mathsf{RandBit}, \mathsf{sid}, \mathsf{id})$ from all honest parties (where id is fresh), send that tuple to the adversary, sample uniform $b \leftarrow \{0, 1\}$, and store (id, b).
- **Linear Combination:** On input $(\mathsf{LinComb}, \mathsf{sid}, \mathsf{id}, (\mathsf{id}_j)_j, c \in R, (c_j \in R)_j)$ from all honest parties (where each id_j is stored in memory and id is fresh), send that tuple to the adversary, retrieve each (id_j, x_j) from memory, compute $y = c + \sum_j c_j \cdot x_j \in R$, and store (id, y).
- **Multiplication:** On input $(\mathsf{Mult}, \mathsf{sid}, \mathsf{id}, \mathsf{id}_1, \mathsf{id}_2)$ from all honest parties (where $\mathsf{id}_1, \mathsf{id}_2$ are stored in memory and id is fresh), send that tuple to the adversary, retrieve $(\mathsf{id}_1, x), (\mathsf{id}_2, y)$ from memory, compute $z = x \cdot y \in R$, and store (id, z).
- **Opening:** On input $(\mathsf{Open}, \mathsf{sid}, \mathsf{id}, l)$ from all honest parties, where $l \leq k$ (and id is present in memory), retrieve (id, y) from memory, let $y' = y \bmod 2^l$, and output $(\mathsf{Open}, \mathsf{sid}, \mathsf{id}, l, y')$ to the adversary and to all the parties.

Figure 1: Arithmetic black box functionality

unique id associated with x.[1] The ids for ephemeral variables are *distinct* (*not* reused) across multiple calls to the same session of a protocol. Note that a public value can be stored in $\mathcal{F}_{\text{ABB}}$ by calling $\mathsf{LinComb}$ with empty sets of identifiers id_j and coefficients c_j, and c equal to the public value.

We overload the operators $+$ and $\cdot$ instead of explicitly calling $\mathcal{F}_{\text{ABB}}$'s $\mathsf{LinComb}$ and Mult commands. For example, $[z]_k = [x]_k \cdot [y]_k + c$ denotes that the values x, y stored in $\mathcal{F}_{\text{ABB}}$ are multiplied using the Mult command, then the constant value c is added using $\mathsf{LinComb}$, and the resulting value z is stored in $\mathcal{F}_{\text{ABB}}$'s memory. This overloading of notation is commonly used in prior works, e.g., [32].

We also extend $\mathcal{F}_{\text{ABB}}$ for the ring $\mathbb{Z}_{2^k}$ to store and operate on values in $\mathbb{Z}_{2^l}$, for any $l < k$. We denote such stored values by $[x]_l$, later opening at most l of their bits, and implicitly "down-cast" stored values modulo smaller powers of two as needed. All this is well defined via the natural (mod-2^l) ring homomorphism from $\mathbb{Z}_{2^k}$ to $\mathbb{Z}_{2^l}$. In other words, we can generically obtain this enhancement by using, in place of $x \in \mathbb{Z}_{2^l}$ itself, any $\bar{x} \in \mathbb{Z}_{2^k}$ for which $\bar{x} \equiv x \pmod{2^l}$. Moreover, in typical realizations of $\mathcal{F}_{\text{ABB}}$ like [19, 28], this can be implemented more directly and efficiently, simply by reducing each share by a correspondingly smaller modulus.

2.2.1 *Realizing $\mathcal{F}_{ABB}$.*

Protocols that realize $\mathcal{F}_{\text{ABB}}$ for various adversary models has been extensively studied in the literature (though more often in the prime-field case, whereas we work with power-of-two modulus).

- **Dishonest majority.** Most commonly, the setting of dishonest majority (with abort) is covered by SPDZ line of work (over $\mathbb{Z}_{2^k}$), e.g., [19, 22, 27, 28]. In this setting, additive secret-sharing is used, and all shares are authenticated via information-theoretic message authentication codes (MACs). This technique is very efficient, as it adds very little overhead compared to a realization for semi-honest adversaries.
- **Honest (super-)majority**. In these settings, it is common to use either replicated secret-sharing (for a small number of parties, due to the size of the shares) or Shamir secret sharing over Galois rings. These techniques provide both semi-honest and active security, and have been explored in several works [20, 28]. In the case of an honest super-majority (i.e., $t < n/3$ corrupt parties), using standard error-correction techniques on the parties' revealed shares can ensure robustness and fairness; see, e.g., [28] for details.

Essentially all realizations of $\mathcal{F}_{\text{ABB}}$ work via some linear secret-sharing scheme, which means that $\mathsf{LinComb}$ can be performed locally (with no communication between parties) using linear operations on shares. By contrast, Mult typically requires some communication, and is often implemented by preprocessing Beaver multiplication triples [3, 14, 25].

2.3 Preprocessed Gates

Our protocols obtain their online efficiency from the offline preparation and careful use of *preprocessed gates*. Let $F\colon A \to B$ be an arbitrary function, where $A = \mathbb{Z}_{2^a}$ and $B = \mathbb{Z}_{2^b}$. A preprocessed gate for F consists of:

1. a stored value $[r]_a$ in $\mathcal{F}_{\text{ABB}}$ of some secret, uniformly random *masking term* $r \in A$, and
2. a *lookup table* $\left([y_{x'}]_b\right)_{x' \in A}$, stored in $\mathcal{F}_{\text{ABB}}$, of all the outputs $y_{x'} = F(x' - r)$ of the shifted function.

Conceptually, the gate can be seen as a kind of naïve function secret sharing of F.

To *apply* such a gate to a stored $[x]_a$, yielding the stored output $[F(x)]_b$, the parties simply:

1. Let $x' = \mathcal{F}_{\text{ABB}}.\mathsf{Open}([x]_a + [r]_a)$. (Here $x' = x + r \in A$ is the "masked" input.)
2. Output $[y_{x'}]_b = [F(x' - r)]_b = [F(x)]_b$. (This just identifies the appropriate stored value in $\mathcal{F}_{\text{ABB}}$ corresponding to x'.)

This uses one call to $\mathcal{F}_{\text{ABB}}.\mathsf{Open}$; the remainder is one invocation of $\mathsf{LinComb}$ (recall that this is typically realized locally, with no communication) and a local computation of the stored output's identifier. Observe that if $r \in A$ is uniformly random and independent of everything else, then this procedure is information-theoretically secure, because the only revealed value is x', which perfectly hides x.

More generally, we also consider gates for *parts* of functions whose domains A are *infinite* additive groups, like $A = \mathbb{Z}$. In this case, the mask value r is from some suitable finite subset of the domain, and the lookup table is $\left([F(x' - r)]_b\right)_{x' \in X'}$ for some suitable finite $X' \subset A$. Applying such a gate to a stored $[x]_a$ is an ad-hoc process, because opening $x' = x + r \in A$ may reveal information about x, and also it might be that $x' \notin X'$. Instead, we will take care

[1]Note that this matches the widespread notation for secret-shared values, and this is by intent: the reader may think of values stored in $\mathcal{F}_{\text{ABB}}$ as being secret-shared among the parties, in a typical realization of $\mathcal{F}_{\text{ABB}}$. However, we do not use any secret-sharing scheme explicitly, but instead abstract it away with $\mathcal{F}_{\text{ABB}}$.

Functionality $\mathcal{F}_{\mathbf{Decrypt}}$

- **Initialize:** Once (Init, sid) is received from each honest party, run $(pk, \mathbf{s}) \leftarrow$ KeyGen, store $\mathbf{s}$, and send (Init, sid, pk) to the adversary and all the parties. Do not respond to any further Init commands for this session.
- **Decrypt:** On input (Decrypt, sid, $\mathbf{c}$) from all the honest parties, where $\mathbf{c}$ is an LWE ciphertext over $\mathbb{Z}_q$, send (Decrypt, sid, $\mathbf{c}$, $\lfloor \langle \mathbf{c}, \mathbf{s} \rangle \rceil_p$) to the adversary and to all the parties.

Figure 2: Functionality for decrypting LWE ciphertexts

Protocol $\Pi_{\mathbf{Decrypt}}$

- On input (Init, sid), the parties run a distributed $\mathcal{F}_{\text{ABB}}$-hybrid protocol for KeyGen that results in a stored secret key $[\mathbf{s}]_k$ and a public key pk.
- On input (Decrypt, sid, $\mathbf{c}$), the parties do the following:
(1) Use $\mathcal{F}_{\text{ABB}}$ to compute $[z]_k = \langle \mathbf{c}, [\mathbf{s}]_k \rangle + 2^{l-1}$.
(2) Run $\text{Mod}_{l,k}(\text{sid}, [z]_k)$ to produce $[e]_k$.
(3) Call $\mathcal{F}_{\text{ABB}}$.Open on $[z]_k - [e]_k$ and receive $\mu' \in \mathbb{Z}_q$ (which is a multiple of L).
(4) Output $\mu = \mu'/L \in \mathbb{Z}_p$.

Figure 3: Protocol for decrypting LWE ciphertext

to open only partial values that fully mask the stored input, and to guarantee that the needed output is stored in the lookup table. Our Sign gates, as constructed in Section 4.2 and used in Section 3.3, are the primary example of this.

3 DISTRIBUTED LWE DECRYPTION VIA MPC ROUNDING

In this section we define an abstract ideal functionality for decrypting LWE ciphertexts—in particular, decryption in all widely used FHE schemes—and give a protocol that uses $\mathcal{F}_{\text{ABB}}$ to perfectly realize this functionality. The functionality is defined in Section 3.1, the protocol and its main subroutine are defined in Sections 3.2 and 3.3, and suggested parameterizations are given in Section 3.4.

3.1 Functionality $\mathcal{F}_{\mathbf{Decrypt}}$

Here we define and discuss our abstract ideal functionality $\mathcal{F}_{\text{Decrypt}}$ (Figure 2) for decrypting LWE ciphertexts. It is parameterized by a power-of-two ciphertext modulus $q = 2^k$ and plaintext modulus $p = 2^m$ for some positive integers $k > m$. These parameters are used globally throughout this work.

The functionality $\mathcal{F}_{\text{Decrypt}}$ has two commands: Init, which generates a public/secret key pair, and Decrypt, which may be called many times to decrypt given ciphertexts. The Decrypt command simply outputs the rounded (from $\mathbb{Z}_q$ to $\mathbb{Z}_p$) inner product of the given ciphertext and the secret key.

Discussion. The functionality is parameterized by a KeyGen algorithm for the underlying LWE-based cryptosystem. The Init command simply runs this algorithm, stores the secret key $\mathbf{s}$ for later use, and outputs the public key to all parties. In our realization we assume a secure $\mathcal{F}_{\text{ABB}}$-aided procedure for KeyGen, which can be obtained generically by standard MPC techniques, or more efficiently using the scheme's specific structure. For example, we can generate public LWE samples for a secret $\mathbf{s}$ by sampling secret random errors and using $\mathcal{F}_{\text{ABB}}$'s linearity features.

We *strongly emphasize* that our $\mathcal{F}_{\text{Decrypt}}$ is not a complete functionality for threshold encryption; it is just a key ingredient in a protocol to realize such a functionality. In order to be meaningfully secure, this protocol would need to be designed so that honest parties Decrypt only ciphertexts that are known to be suitably "well formed." Otherwise, the Decrypt command would act as an unrestricted decryption oracle, from which it is easy to learn the secret key using standard techniques. Specific restrictions on decrypted ciphertexts in the threshold setting have been considered in many prior works [20, 33] , and are outside the scope of this work.

3.2 Protocol $\Pi_{\mathbf{Decrypt}}$

Here we define our main decryption protocol (Figure 3), which perfectly realizes $\mathcal{F}_{\text{Decrypt}}$ in the $\mathcal{F}_{\text{ABB}}$-hybrid model. (See Section 2.2 for the details of $\mathcal{F}_{\text{ABB}}$.) Recalling that the ciphertext modulus is $q = 2^k$ and the plaintext modulus is $p = 2^m < q$, we let $l = k - m \geq 1$ and $L = 2^l = q/p$.

For Decrypt on a given ciphertext $\mathbf{c}$, the parties use the stored secret key to linearly compute (inside $\mathcal{F}_{\text{ABB}}$) the "noisy decryption value" $[z]_k = [\mu \cdot L + e]_k$, where the message $\mu \in \mathbb{Z}_{2^m}$ and the noise $e \in [L]$ (for convenience, we shift the noise term to be non-negative). In other words, the message occupies the high m bits, and the noise occupies the low l bits. Then, using the $\text{Mod}_{l,k}$ subroutine (Procedure 1), the parties securely compute and cancel out the noise term, i.e., they compute $[e]_k = [z \bmod L]_k$ and $[z]_k - [e]_k = [\mu \cdot L]_k$. Finally, they Open the latter value to get μ.[2]

The main challenge, therefore, is to securely compute the noise term $[e]_k = [z \bmod L]_k$ from $[z]_k$, i.e., mod-L reduction on a k-bit secret input. We give a subroutine for this in Section 3.3 below. The following is our main security theorem, which follows straightforwardly from the security and correctness claims for our subroutines below.

Theorem 1. *Protocol $\Pi_{Decrypt}$ (Figure 3) perfectly realizes the ideal functionality $\mathcal{F}_{Decrypt}$ in the $\mathcal{F}_{ABB}$-hybrid model (where $\mathcal{F}_{Decrypt}$ and $\mathcal{F}_{ABB}$ have the same "shell," i.e., adversary corruption model).*

The proof of Theorem 1 is given in Appendix A.

3.3 Secure Mod and Comparison via $\text{LTRand}_{l,k}$

Here we securely implement the $\text{Mod}_{l,k}$ subroutine and similar comparison functions, like (modular) less-than-zero ModLTZ_k. We implement these as "thin wrappers" around a new core abstraction

[2]We remark that opening the original value z itself would reveal the noise in the ciphertext, which cannot be simulated from the decrypted message alone, and typically can lead to a complete break of the scheme.

and efficient procedure we call $\mathrm{LTRand}_{l,k}$, which is one of our main technical contributions.

Essentially, $\mathrm{LTRand}_{l,k}$ takes a stored input in $\mathbb{Z}_{2^k}$ (or $[2^k]$), randomly masks it modulo 2^l, and returns the (opened) masked value, the stored mask, and a stored bit indicating whether the former is less than the latter. Its precise specification is as follows, and our efficient implementation is given below in Procedure 3.

Input: stored value $[z]_k$ for some $z \in \mathbb{Z}_{2^k}$.

Output:
- $z' = z + r \bmod 2^l$ for fresh uniformly random $r \in [L]$,
- stored value $[r]_k$, and
- stored bit $[u]_k$, where $u = (z' \overset{?}{<} r) = (z' - r \overset{?}{<} 0) \in \{0, 1\}$.

Observe that the output reveals nothing, because $z' \in [2^l]$ is uniformly random for any input z.

Although the interface of $\mathrm{LTRand}_{l,k}$ may seem somewhat ad-hoc, it is rich enough to directly implement several fundamental comparison procedures of wide applicability, even beyond our immediate purposes, e.g., secure machine learning [34, 43, 48, 50]. We next give two useful examples of this.

Secure mod and comparison. $\mathrm{Mod}_{l,k}$, as defined in Procedure 1, securely implements modular reduction, mapping a stored input $[z]_k$ to stored output $[z \bmod 2^l]_k$. Importantly, the output is also a k-bit value in $\mathbb{Z}_{2^k}$, with zeros in its most significant $k - l$ bits.[3] (Recall that in the decryption protocol Π_{Decrypt}, this is needed to extract and cancel out the ciphertext noise, leaving the message unaffected.)

As an optimization, with a typical realization of $\mathcal{F}_{\mathrm{ABB}}$ it suffices for $\mathrm{LTRand}_{l,k}$ to produce $[u]_{k-l}$ instead of $[u]_k$, which can save significant storage in our typical setting where $k - l \ll k$. This is because $2^l \cdot [u]_{k-l}$ can naturally be treated as $[2^l \cdot u]_k$ in the underlying secret-sharing schemes.

Procedure 1: $\mathrm{Mod}_{l,k}(\mathsf{sid}, [z]_k)$

(1) Run $\mathrm{LTRand}_{l,k}(\mathsf{sid}, [z]_k)$ to get z', $[r]_k$, $[u]_k$.
(2) Output $[e]_k = z' - [r]_k + 2^l \cdot [u]_k$.

Lemma 2. *Procedure $\mathrm{Mod}_{l,k}$ is correct and secure.*

Proof. Security follows immediately from that of $\mathrm{LTRand}_{l,k}$. For correctness, the output $e = z' - r + 2^l \cdot u$ satisfies $e \equiv z' - r \equiv z \pmod{2^l}$ and $e \in [2^l]$, because $z' - r \in (-2^l, 2^l)$ (since $z', r \in [2^l]$) and $u = (z' - r \overset{?}{<} 0)$. So, $e = z \bmod 2^l$, as claimed. □

As another example, for any integer $t \geq 1$ defining $T = 2^t$, define the "modular less-than-zero" function $\mathrm{ModLTZ}_t \colon \mathbb{Z}_T \to \{0, 1\}$ as

$$\mathrm{ModLTZ}_t(x) := \begin{cases} 0 & \text{if } x \in [0, T/2) \pmod{T} \\ 1 & \text{if } x \in [-T/2, 0) \equiv [T/2, T) \pmod{T}. \end{cases} \tag{1}$$

In other words, this corresponds to whether the input's *signed* representative in $[-T/2, T/2)$ is less than zero; equivalently, it is the most-significant bit of the input's t-bit *unsigned* representative in $[0, T)$. Procedure 2 gives a secure procedure for this function; it is closely inspired by the MSB procedure in [22].

Procedure 2: $\mathrm{ModLTZ}_t(\mathsf{sid}, [z]_t)$

(1) Run $\mathrm{Mod}_{t-1,t}(\mathsf{sid}, [z]_t)$ to get $[e]_t = [z \bmod T/2]_t$.
(2) Call $\mathcal{F}_{\mathrm{ABB}}$.RandBit to get $[b]_t$.
(3) Call $\mathcal{F}_{\mathrm{ABB}}$.Open on $[z]_t - [e]_t + (T/2) \cdot [b]_t$, yielding $(T/2) \cdot h' \in \mathbb{Z}_T$ for some $h' \in \{0, 1\}$.
(4) Output $h' + [b]_t - 2h' \cdot [b]_t$.

Lemma 3. *Procedure ModLTZ_t is correct and secure.*

Proof. By Lemma 2, $e = z \bmod T/2$, so $z - e \equiv (T/2) \cdot h \pmod{T}$, where $h = \mathrm{ModLTZ}_t(z) \in \{0, 1\}$. Therefore, the opened bit $h' = h \oplus b$ reveals nothing (it is uniformly random, for any z). Finally, $h + b - 2h'b = h' \oplus b = h = \mathrm{ModLTZ}_t(z)$, as desired. □

Implementation of $\mathrm{LTRand}_{l,k}$. The $\mathrm{LTRand}_{l,k}$ procedure has an offline-online structure with a very efficient online phase, thanks to its usage of rich preprocessed gates, as prepared in the offline phase. It is parameterized by an input bit length b for (most of) these gates, which determines the following additional parameters:

- For $L = 2^l$, values in $[L]$ are represented in base $B = 2^b$, i.e., using "digits" of bit length b.
- The number of digits is therefore $d = \lceil l/b \rceil$, where we let $D = 2^d$, and the bit length of the most-significant digit is $b' = l - (d-1)b \leq b$, where we let $B' = 2^{b'}$.

In the offline phase, $\mathrm{LTRand}_{l,k}$ prepares (using the procedures given in Section 4) several Sign gates for b-bit inputs, and one ModLTZ gate for a $(d+1)$-bit input. Each of these is used just once in a later call to the online phase.

The online phase first opens $z' = z + r \bmod L$, where $r \in [L]$ is a uniformly random mask constructed from the masks r_i of the individual Sign gates—specifically, the r_i are the base-B digits of r. (Observe that r perfectly masks the low l bits of z, so z' reveals nothing.) It then uses the preprocessed gates to securely compute (inside $\mathcal{F}_{\mathrm{ABB}}$) the bit

$$u = (z' \overset{?}{<} r) \in \{0, 1\}.$$

Finally, it outputs z' and the stored values r, u.

The main challenge lies with securely computing the bit u. The remainder of this subsection describes how we do this using the preprocessed Sign and ModLTZ gates.

Define the function $\mathrm{Sign} \colon \mathbb{Z} \to \{-1, 0, 1\}$ as

$$\mathrm{Sign}(x) := \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ +1 & \text{if } x > 0. \end{cases} \tag{2}$$

Our approach is based on the following central recurrence for the Sign function.

Lemma 4. *Let $x, r \in [B^d] = [2^{bd}]$ have base-B representations $x = \sum_{i \in [d]} x_i \cdot B^i$ with each $x_i \in [B]$, and similarly for r. Then the*

[3]Recall that reducing from modulus 2^k to 2^l for $l < k$ is trivially supported by $\mathcal{F}_{\mathrm{ABB}}$, but the result merely has l bits, whereas here we need to retain k total bits of precision, with zeros in the high bits.

sign of $x - r$ is the "sign of weighted signs" of the digit differences:

$$Sign(x-r) = Sign\Big(\sum_{i\in[d]} (x_i-r_i)\cdot B^i\Big) = Sign\Big(\sum_{i\in[d]} Sign(x_i-r_i)\cdot 2^i\Big). \tag{3}$$

Proof. If $x_{d-1} - r_{d-1} \neq 0$, then $\text{Sign}(x-r) = \text{Sign}(x_{d-1} - r_{d-1})$, since

$$|x_{d-1} - r_{d-1}| \cdot B^{d-1} > \Big|\sum_{i\in[d-1]} (x_i - r_i)\cdot B^i\Big|$$

by the triangle inequality, the fact that each $|x_i - r_i| \leq B - 1$, and geometric sums. For similar reasons, $\text{Sign}(x_{d-1} - r_{d-1})$ equals the right-hand side of Equation (3), because each $|\text{Sign}(x_i - r_i)| \leq 1$. Otherwise, $x_{d-1} - r_{d-1} = 0$, so its contribution to each side of Equation (3) is zero, and the claim holds by induction. □

Similarly to Lemma 4, for any $x, r \in [B^d]$ with their d-digit base-B representations,

$$(x \overset{?}{<} r) = (x-r \overset{?}{<} 0) = \text{ModLTZ}_{d+1}\Big(\sum_{i\in[d]} \text{Sign}(x_i-r_i)\cdot 2^i \bmod 2D\Big). \tag{4}$$

We emphasize that the weighted sum is in the interval $(-D, D)$, so the reduction modulo $2D$ does not "wrap around," hence the equation holds true. We need this form of the equation because in our procedure, the weighted sum is computed by $\mathcal{F}_{\text{ABB}}$, which is limited to modular arithmetic with a certain amount of precision.

Altogether, Equation (4) gives a very efficient way of securely computing the bit $u = (z' \overset{?}{<} r)$ using the preprocessed Sign gates and ModLTZ gate. Specifically, using the digits z'_i of z', we first look up the stored digit-difference signs $[\text{Sign}(z'_i - r_i)]$ from the Sign gates, then compute their weighted sum, then apply the ModLTZ gate to the result.

Procedure 3: $\text{LTRand}_{l,k}(\text{sid}, [z]_k)$

Preprocessing phase (before $[z]_k$ is known):

(1) Call $\text{PrepModLTZ}_{d+1,m}(\text{sid})$ (Procedure 7) to prepare a ModLTZ gate.
(2) For each $i \in [d-1]$, call $\text{PrepSign}_{b,d+1}(\text{sid})$ (Procedure 5) to prepare a Sign gate
$$[r_i]_k\ ,\ ([\text{Sign}(x_i - r_i)]_{d+1})_{x_i\in[B]}\ .$$
For $i = d - 1$, do the same but with b', B' in place of b, B (respectively).
(3) Compute $[r]_k = \sum_{i\in[d]} [r_i]_k \cdot B^i$.
▷ $r = (r_{d-1} \cdots r_1 r_0)_B$ is the base-B representation of a uniformly random $r \in [L]$.

Online phase (once $[z]_k$ is known):

(1) Call $\mathcal{F}_{\text{ABB}}$.Open on $[z]_l + [r]_l$ and receive $z' \in [L]$.
▷ $z' = z + r \bmod L$
(2) Express z' in base B, as $z' = \sum_{i\in[d]} z'_i \cdot B^i$ where $z'_i \in [B]$, and $z'_{d-1} \in [B']$.
(3) For each $i \in [d]$, let $[y_i]_{d+1} = [\text{Sign}(z'_i - r_i)]_{d+1}$ from the corresponding Sign gate.
(4) Compute $[y]_{d+1} = \sum_{i\in[d]} [y_i]_{d+1} \cdot 2^i$.
(5) Apply (as defined in Section 2.3) the ModLTZ gate to $[y]_{d+1}$, yielding $[u]_k$. ▷ $u = (z' \overset{?}{<} r)$, by Equation (4)
(6) Output z', $[r]_k$, $[u]_k$.

Lemma 5. *Procedure $LTRand_{l,k}$ is correct and secure.*

Proof. For security, the only opened value is $z' = z + r \bmod L$, which reveals nothing because it is uniformly random in $[L]$ for any input z. For correctness, adopting the procedure's notation, we analyze the values that are computed and opened by $\mathcal{F}_{\text{ABB}}$. First, on Line (1), $\mathcal{F}_{\text{ABB}}$ opens $z' = z + r \bmod L$, where $r \in [L]$ is uniformly random. Then, by the correctness of the Sign gates, Lines (3) and (4) compute $y = \sum_{i\in[d]} \text{Sign}(z'_i - r_i) \cdot 2^i \bmod 2D$. Then, by Equation (4) and the correctness of the ModLTZ gate, Line (5) computes $u = (z' \overset{?}{<} r)$. □

3.4 Efficiency and Parameters

Communication and rounds. The online phase of the protocol has very low communication, and few rounds. Specifically, each invocation of Decrypt has three sequential calls to $\mathcal{F}_{\text{ABB}}$.Open:

(1) at the start of Mod, to reveal a masked l-bit value;
(2) when applying the ModLTZ gate, to reveal a masked $(d + 1)$-bit value; and
(3) when opening the decrypted message, to reveal an unmasked k-bit value.

So, the total number of opened bits is $l + (d + 1) + k$. The rest of the online phase is either local computation or calls to $\mathcal{F}_{\text{ABB}}$.LinComb, which in a typical realization are also just local computation (There are no calls to $\mathcal{F}_{\text{ABB}}$.Mult§).

Gate storage. Each call to Decrypt consumes, and hence requires producing, several preprocessed gates in $\mathcal{F}_{\text{ABB}}$. Each invocation of Decrypt uses (ignoring the masking values, which are insignificant):

- One ModLTZ gate, represented by a table of 2^{d+1} stored m-bit values (using the optimization mentioned in Section 3.3).
- $d = \lceil l/b \rceil$ Sign gates, each represented by a table of $B = 2^b$ (or $B' = 2^{b'}$ for the most-significant digit) stored $(d+1)$-bit values.

The total bit length of the stored gates (again, ignoring masks) is therefore

$$(d+1) \cdot ((d-1) \cdot 2^b + 2^{b'}) + m \cdot 2^{d+1}.$$

The above is approximately optimized for $b \approx d \approx \sqrt{l} = \sqrt{k-m}$, though finer-grained optimization tends to produce somewhat larger d and somewhat smaller b.

Realization with SPD$\mathbb{Z}_{2^k}$. We give some concrete communication and storage numbers when $\mathcal{F}_{\text{ABB}}$ is instantiated naïvely using the SPD$\mathbb{Z}_{2^k}$ protocol [19], for the realistic example parameters

$$k = 64\,,\ m = 1\,,\ b = 8 \text{ and hence } d = 8\,,\ b' = 7.$$

We use $s = 64$ for the MAC's statistical security parameter.

Naïvely, the online phase has four additional rounds, for a total of seven real rounds of communication. However, we believe that this can be optimized, because the first two Open rounds are of randomly masked values. So, they are analogous to the masked values that are opened during Mult commands, whose MAC checks can be batched and deferred to just before opening an unmasked value.

Here, each party should add s extra bits for masking the value. It also requires each party to commit to and reveal a single (deferred batch) MAC of $k+s$ bits. So, the total communication per party is just 712 bits, plus the commitments. This can potentially be reduced even further using the above-described optimization.

For the gate storage per invocation of Decrypt, $\mathcal{F}_{\text{ABB}}$ stores and consumes a total of 17 792 bits. In the SPD$\mathbb{Z}_{2^k}$ realization of $\mathcal{F}_{\text{ABB}}$, each stored t-bit value has a share size of $t+s$ bits, and an associated MAC share size of $t+s$ bits as well. So, each party stores a total of 346 880 bits.

4 PREPROCESSING PROCEDURES

Here we show how to efficiently prepare gates—i.e., random mask values r and corresponding lookup tables—for the Sign and ModLTZ functions. A key commonality is that, as we show, the lookup tables for both functions can be seen as *"structured" linear transforms* of the *subset-products* of the bits of r. These structured transforms can be evaluated by fast algorithms, roughly analogous to the Fast Fourier Transform. In Section 4.1 we first give a simple protocol for computing a random mask value along with the subset-products of its bits. Then, in Sections 4.2 and 4.3 we derive the fast linear transforms that map from the subset-product vectors to the lookup tables.

4.1 Procedure PrepSubsetProds

Let $a \geq 1$ be an input length (in bits) with $A = 2^a$, and $c \leq k$ be an output length. Here we give a procedure, in the $\mathcal{F}_{\text{ABB}}$-hybrid model, that prepares a uniformly random $r \in [A]$ and all the subset-products of its bits, as integers modulo 2^c.

In what follows, it is convenient to index vectors by subsets. For this purpose, the subset $S \subseteq [a]$ corresponds to the index

$$\sum_{s \in S} 2^s = \sum_{i \in [a]} \delta_{S,i} \cdot 2^i \in [A]\,,$$

where $\delta_{S,i}$ is 1 if $i \in S$, and 0 otherwise.

Procedure 4: PrepSubsetProds$_{a,c}$(sid)

Output: $[r]_k$ for a uniformly random $r = (r_{a-1} \cdots r_0)_2 \in [A]$, along with the vector $\big([p_S]_c\big)_{S \subseteq [a]}$ of all the subset-products $p_S = \prod_{i \in S} r_i \in \{0,1\}$ of the bits r_i of r.

(1) If $a = 1$, call $\mathcal{F}_{\text{ABB}}$ on (RandBit, sid) to generate $[r]_k$, and output $[r]_k, ([1]_c, [r]_c)$.
(2) Write $a = a_0 + a_1$ for some $a_0, a_1 \geq 1$ (typically, $a_0 = \lceil a/2 \rceil$). In parallel, recursively call PrepSubsetProds$_{a_0,c}$(sid) and PrepSubsetProds$_{a_1,c}$(sid) to prepare (respectively)

$$[r_0]_k\,,\ \big([p_{S_0}]_c\big)_{S_0 \subseteq [a_0]} \quad \text{and} \quad [r_1]_k\,,\ \big([p'_{S_1}]_c\big)_{S_1 \subseteq [a_1]}\,.$$

(3) Let $[r]_k = [r_0]_k + 2^{a_0} \cdot [r_1]_k$.
(4) For each $S_0 \subseteq [a_0]$ and nonempty $S_1 \subseteq [a_1]$, use $\mathcal{F}_{\text{ABB}}$ to compute $[p_{S_0 \cup (a_0 + S_1)}]_c = [p_{S_0}]_c \cdot [p'_{S_1}]_c$.
(5) Output $[r]_k, \big([p_S]_c\big)_{S \subseteq [a]}$.

Lemma 6. *Procedure PrepSubsetProds$_{a,c}$ is correct.*

Proof. The base case is correct by inspection. For the recursive case, first note that $r = r_0 + 2^{a_0} \cdot r_1 \in [A] = [2^a]$ is uniformly random because $r_0 \in [2^{a_0}], r_1 \in [2^{a_1}]$ are uniform and independent, by induction. For correctness of the subset-products, observe that every subset $S \subseteq [a]$ is the disjoint union of some $S_0 \subseteq [a_0]$ and the shifted subset $a_0 + S_1$ for some $S_1 \subseteq [a_1]$. By induction, p_{S_0}, p'_{S_1} are the subset-products of the bits of r_0, r_1 indexed by S_0, S_1 (respectively), so $p_{S_0} \cdot p'_{S_1}$ is the subset-product of the bits of $r = r_0 + 2^{a_0} \cdot r_1$ indexed by $S = S_0 \cup (a_0 + S_1)$. □

Efficiency analysis. The procedure can implemented using a total of $2^a - a - 1$ calls to $\mathcal{F}_{\text{ABB}}$.Mult. For the base case this is true by inspection. For the recursive case, a call to Mult is needed only for *nonempty* S_0, S_1, because the empty set has subset-product 1. So, the number of Mult calls satisfies the recurrence $M(a) = M(a_0) + M(a_1) + (2^{a_0} - 1)(2^{a_1} - 1)$, which solves to $M(a) = 2^a - a - 1$ by induction.

The number of sequential "rounds" of (parallel) Mult calls can be as small as $\lceil \log_2 a \rceil$, by always taking $a_0 = \lceil a/2 \rceil$.

4.2 Procedure PrepSign

The procedure PrepSign$_{b,d+1}$ (Procedure 5) prepares a Sign gate for b-bit inputs, i.e., the domain $[B]$ where $B = 2^b$, with output modulo $2D = 2^{d+1}$. It does this by first preparing the subset-products of the bits of a (secret) uniformly random $r \in [B]$, then converts those subset-products to the vector of Sign$(x - r)$ values for all $x \in [B]$. The conversion is a *linear* function with a *fast* implementation via a recursive divide-and-conquer procedure, which we give below in Procedure 6.

Procedure 5: $\mathrm{PrepSign}_{b,d+1}(\mathsf{sid})$

(1) Call $\mathrm{PrepSubsetProds}_{b,d+1}(\mathsf{sid})$ to prepare $[r]_k$, $[\vec{p}]_{d+1}$
(2) Output $[r]_k$, $\mathrm{SubsetProdsToSigns}_b([\vec{p}]_{d+1})$ ▷ Procedure 6

Fast linear transform from subset-products to signs. Here we derive the fast linear transform that converts the subset-products of the bits of r to the values of $\mathrm{Sign}(x-r)$, for all $x \in [B]$. The key idea is that the output vector satisfies a simple recurrence relation in terms of the top bits of x and r (respectively), and the output vector for their remaining $b-1$ bits each; see Equation (5) below. So, there is a fast algorithm that recursively evaluates the transform, analogous to the Fast Fourier Transform.

Because the transform is linear, the algorithm can be implemented straightforwardly on values stored in $\mathcal{F}_{\mathrm{ABB}}$ using just its LinComb operation, and hence using just local computation (no communication) in a typical realization of $\mathcal{F}_{\mathrm{ABB}}$. Therefore, for simplicity of presentation we omit the $[\cdot]$ notation around all the values, and present the algorithm as operating on the values themselves.

View $x, r \in [B]$ in binary as $x = (x_{b-1}\cdots x_0)_2$ and $r = (r_{b-1}\cdots r_0)_2$. When $b = 1$, clearly $\mathrm{Sign}(x-r) = x_0 - r_0$. When $b > 1$, the key insight is that for the "truncated" values $x' = (x_{b-2}\cdots x_0)_2$ and $r' = (r_{b-2}\cdots r_0)_2$, we have the recurrence relation

$$\mathrm{Sign}(x-r) = (x_{b-1}-r_{b-1}) + \mathrm{Sign}(x'-r') \cdot \begin{cases} 1-r_{b-1} & \text{if } x_{b-1} = 0 \\ r_{b-1} & \text{if } x_{b-1} = 1. \end{cases} \tag{5}$$

This can be seen by observing that if $x_{b-1} \neq r_{b-1}$, $\mathrm{Sign}(x-r) = x_{b-1} - r_{b-1}$ (the less-significant bits are irrelevant); otherwise, $x_{b-1} - r_{b-1} = 0$ and hence $\mathrm{Sign}(x-r) = \mathrm{Sign}(x'-r')$. In the former case, $\mathrm{Sign}(x'-r')$ is multiplied by 0 in the above expression, and in the latter case it is multiplied by 1.

Procedure 6: Linear Function $\mathrm{SubsetProdsToSigns}_b(\vec{p})$

Input: $\vec{p} = (p_S)_{S \subseteq [b]}$, where $p_S = \prod_{i \in S} r_i$ for some $r = (r_{b-1}\cdots r_0)_2 \in [B]$
Output: vector $\big(\mathrm{Sign}(x-r)\big)_{x \in [B]}$
(1) If $b = 0$ output (0).
(2) Split $\vec{p}$ into its initial half $\vec{p}^{(0)} = (p_{S'})_{S' \subseteq [b-1]}$ and latter half $\vec{p}^{(1)} = r_{b-1} \cdot \vec{p}^{(0)}$.
(3) Apply $\mathrm{SubsetProdsToSigns}_{b-1}$ to each half, yielding
$$\vec{s}^{(0)} = (\mathrm{Sign}(x'-r'))_{x' \in [B/2]}, \quad \vec{s}^{(1)} = r_{b-1} \cdot \vec{s}^{(0)}$$
respectively, where $r' = (r_{b-2}\cdots r_0)_2$.
(4) Output the vector, indexed by $[B]$, whose initial half is
$$\begin{aligned} &-p_\emptyset^{(1)} \cdot \vec{1}_{B/2} + (\vec{s}^{(0)} - \vec{s}^{(1)}) \\ &\quad = -r_{b-1} \cdot \vec{1}_{B/2} + \vec{s}^{(0)} \cdot (1 - r_{b-1}) \end{aligned}$$
and whose latter half is
$$\begin{aligned} &(p_\emptyset^{(0)} - p_\emptyset^{(1)}) \cdot \vec{1}_{B/2} + \vec{s}^{(1)} \\ &\quad = (1 - r_{b-1}) \cdot \vec{1}_{B/2} + \vec{s}^{(0)} \cdot r_{b-1}, \end{aligned}$$
where $\vec{1}_{B/2}$ is the all-ones vector of dimension $B/2$.

The algorithm is correct by its correct recursive computation of $\vec{s}^{(0)}$ and $\vec{s}^{(1)}$, which follows from its linearity and the relationship between $\vec{p}^{(0)}$ and $\vec{p}^{(1)}$, and by Equation (5). The running time is $O(B \log B) = O(Bb)$ additions and subtractions, due to the recursive divide-and-conquer nature of the algorithm.

4.3 Procedure PrepModLTZ

The procedure $\mathrm{PrepModLTZ}_{d+1,m}$ (Procedure 7) prepares a ModLTZ gate for $(d+1)$-bit inputs, i.e., the domain $\mathbb{Z}_{2D}$ where $D = 2^d$, with output modulo 2^m. It does this by first preparing the subset-products of the bits of a (secret) uniformly random $r \in \mathbb{Z}_{2D}$, then converts those subset-products to the vector of $\mathrm{ModLTZ}_{d+1}(x-r)$ values for all $x \in \mathbb{Z}_{2D}$. As with PrepSign, the conversion is a *linear* function with a fast divide-and-conquer implementation, which we give below in Procedure 8.

Procedure 7: $\mathrm{PrepModLTZ}_{d+1,m}(\mathsf{sid})$

(1) Call $\mathrm{PrepSubsetProds}_{d+1,m}(\mathsf{sid})$ to prepare $[r]_k$, $[\vec{p}]_m$
(2) Output $[r]_k$, $\mathrm{SubsetProdsToModLTZ}_{d+1}([\vec{p}]_m)$ ▷ Procedure 8

Fast linear transform from subset-products. Here we derive a fast linear transform that converts the subset-products of the bits of $r \in \mathbb{Z}_{2D}$ to the values of $\mathrm{ModLTZ}_{d+1}(x-r)$ for all $x \in \mathbb{Z}_{2D}$. As above, we give a recurrence relation in terms of the top bits of x and r (respectively) and their remainders; see Equations (6) and (7) below. This in turn yields a fast recursive algorithm that evaluates the transform. As above, because the transform is linear, we express it as operating on vectors of values themselves, omitting the $[\cdot]$ notation.

First, for any $a \geq 1$ and integers $x, \bar{r} \in [2^a]$, define

$$\mathrm{Carry}_a(x, \bar{r}) := (x + \bar{r} + 1 \overset{?}{\geq} 2^a)$$

to be the "carry" (or "overflow") bit of $x + \bar{r} + 1$. For convenience, we also define the trivial base case $\mathrm{Carry}_0(\varepsilon, \varepsilon) = 1$. Then by inspection, this function satisfies the recurrence

$$\mathrm{Carry}_a(x, \bar{r}) = \begin{cases} \bar{r}_{a-1} \cdot \mathrm{Carry}_{a-1}(x', \bar{r}') & \text{if } x_{a-1} = 0 \\ \bar{r}_{a-1} + (1 - \bar{r}_{a-1}) \cdot \mathrm{Carry}_{a-1}(x', \bar{r}') & \text{if } x_{a-1} = 1, \end{cases} \tag{6}$$

where $x = x_{a-1} \cdot 2^{a-1} + x'$ for its top bit $x_{a-1} \in \{0,1\}$ and remainder $x' \in [2^{a-1}]$, and similarly for $\bar{r}, \bar{r}_{a-1}, \bar{r}'$. This recurrence directly yields the function SubsetProdsToCarries in Procedure 9 below.

Now we turn to $\mathrm{ModLTZ}_{d+1}(x-r)$. Identify $x, r \in \mathbb{Z}_{2D}$ with their $(d+1)$-bit representatives in $[2D]$, and recall from Equation (1) that $\mathrm{ModLTZ}_{d+1}(x-r)$ outputs whether $x - r \in [-D, 0) \equiv [D, 2D) \pmod{2D}$. This is equivalent to the dth bit (counting from 0) of $x + \bar{r} + 1$, where $\bar{r} = (2D-1) - r \in [2D]$ is the integer represented by the bitwise complement of r. So by inspection,

$$\mathrm{ModLTZ}_{d+1}(x-r) = \begin{cases} \bar{r}_d + (1 - 2\bar{r}_d) \cdot \mathrm{Carry}_d(x', \bar{r}') & \text{if } x_d = 0 \\ (1 - \bar{r}_d) + (2\bar{r}_d - 1) \cdot \mathrm{Carry}_d(x', \bar{r}') & \text{if } x_d = 1, \end{cases} \tag{7}$$

where $x = x_d \cdot 2^d + x'$ for its top bit $x_d \in \{0,1\}$ and remainder $x' \in [2^d]$, and similarly for $\bar{r}, \bar{r}_d, \bar{r}'$. This equation directly yields the function SubsetProdsToModLTZ in Procedure 8 below. Similarly to the above, its running time is $O(D \log D) = O(Dd)$ additions and subtractions.

Procedure 8: Linear Function $\mathrm{SubsetProdsToModLTZ}_{d+1}(\vec{p})$

Input: $\vec{p} = (p_S)_{S \subseteq [d+1]}$, where $p_S = \prod_{i \in S} r_i$ for some $r = (r_d \cdots r_0)_2 \in \mathbb{Z}_{2D}$ with $D = 2^d$.

Output: vector $\left(\mathrm{ModLTZ}_{d+1}(x - r)\right)_{x \in \mathbb{Z}_{2D}}$

(1) Split the input vector into its initial half $\vec{p}^{(0)} = (p_{S'})_{S' \subseteq [d]}$ and latter half $\vec{p}^{(1)} = r_d \cdot \vec{p}^{(0)}$.

(2) Apply $\mathrm{SubsetProdsToCarries}_d$ (Procedure 9) to each half, yielding

$$\vec{c}^{(0)} = (\mathrm{Carry}(x', \bar{r}'))_{x' \in [D]} \ , \ \vec{c}^{(1)} = r_d \cdot \vec{c}^{(0)}$$

respectively, where $\bar{r}' = (D-1) - (r_{d-1} \cdots r_0)_2 \in [D]$.

(3) Output the vector, indexed by $\mathbb{Z}_{2D}$ whose initial half (indexed by $[D]$) is

$$\begin{aligned}(p_{\emptyset}^{(0)} - p_{\emptyset}^{(1)}) \cdot \vec{1}_D - \vec{c}^{(0)} + 2\vec{c}^{(1)} \\ = (1 - r_d) \cdot \vec{1}_D + (2r_d - 1) \cdot \vec{c}^{(0)},\end{aligned}$$

and whose latter half is

$$p_{\emptyset}^{(1)} \cdot \vec{1}_D + \vec{c}^{(0)} - 2\vec{c}^{(1)} = r_d \cdot \vec{1}_D + (1 - 2r_d) \cdot \vec{c}^{(0)}.$$

Procedure 9: Linear Function $\mathrm{SubsetProdsToCarries}_d(\vec{p})$

Input: $\vec{p} = (p_S)_{S \subseteq [d]}$, where $p_S = \prod_{i \in S} r_i$ for some $r = (r_{d-1} \cdots r_0)_2 \in [D]$ with $D = 2^d$.

Output: vector $\left(\mathrm{Carry}_d(x, \bar{r})\right)_{x \in [D]}$, where $\bar{r} = (D-1) - r$.

(1) If $d = 0$, output $\vec{p} = (p_{\emptyset}) = (1)$.

(2) Split the input vector into its initial half $\vec{p}^{(0)} = (p_{S'})_{S' \subseteq [d-1]}$ and latter half $\vec{p}^{(1)} = r_{d-1} \cdot \vec{p}^{(0)}$.

(3) Apply $\mathrm{SubsetProdsToCarries}_{d-1}$ to each half, yielding

$$\vec{c}^{(0)} = ([\mathrm{Carry}(x', \bar{r}')])_{x' \in [D/2]} \ , \ \vec{c}^{(1)} = r_{d-1} \cdot \vec{c}^{(0)}$$

respectively, where $\bar{r}' = (D/2 - 1) - (r_{d-2} \cdots r_0)_2$.

(4) Output the vector of shares, indexed by $[D]$, whose initial half is

$$\vec{c}^{(0)} - \vec{c}^{(1)} = (1 - r_{d-1}) \cdot \vec{c}^{(0)},$$

and whose latter half is

$$\begin{aligned}(p_{\emptyset}^{(0)} - p_{\emptyset}^{(1)}) \cdot \vec{1}_{D/2} + \vec{c}^{(1)} \\ = (1 - r_{d-1}) \cdot \vec{1}_{D/2} + r_{d-1} \cdot \vec{c}^{(0)}.\end{aligned}$$

5 IMPLEMENTATION AND EVALUATION

For empirical evaluation purposes, we chose to instantiate $\mathcal{F}_{\mathrm{ABB}}$ in the dishonest-majority setting, which leverages the SPD$\mathbb{Z}_{2^k}$ protocol [19]. As a reminder, this protocol uses authenticated additive secret-sharing.

We implemented[4] the offline and online parts of our protocols separately:

(1) **Offline phase.** Implemented using the well-known MP-SPDZ library [31]. In this part, all Sign and ModLTZ gates are prepared, as described in Section 4.

(2) **Online phase.** Implemented in Rust. Each party executes a program to handle the computation and communication tasks during the protocol's steps. Precomputed Sign and ModLTZ are loaded into memory at the start. To authenticate results, the BatchCheck procedure of SPD$\mathbb{Z}_{2^k}$ [19] is used.

We ran all benchmarks on a single AWS `c7a.32xlarge` instance with 128 AMD EPYC 9R14 vCPUs and 256 GiB of RAM. For the online phase, the Linux tc utility is used to emulate network conditions and constraints. This utility allows precise control over network behavior, such as introducing configurable delays and bandwidth limitations. The selected network has a fully connected topology of nodes using TCP for pairwise communication to execute the protocol.

To compare with the state-of-the-art [20], we used the same LWE parameters that they reported using $(n, q, p) = (1024, 2^{64}, 2)$, and which are also commonly used parameters for TFHE [15] in commercial implementations [5]. As mentioned in Section 3.4, a good selection for the digit bit-length is $b \approx d \approx \sqrt{k - m} \approx 8$, which we use throughout.

It is important to note that [20] did not provide a public implementation we could directly benchmark against and that they only reported their results. Therefore, we ran our experiments in a similar, though not identical, system. They also did not report direct throughput results, but their protocol is CPU-bound by the squish-and-squash operation, which is needed for noise-flooding. For that reason, we assume they can only perform this computation serially on a single machine, and any parallelization would require scaling horizontally. Table 1 shows a benchmark for 4 parties with a network ping time of 1ms and 1Gbit bandwidth. We estimate that our protocol is approximately 37× faster in terms of latency and has about 20,000× higher throughput under these settings.

Protocol	Latency (ms)	Throughput (Dec/s)
[20]	315.62	3.18
Ours	8.48	64 319

Table 1: Comparison of Latency and Throughput for 4 parties with 1ms Ping Time and 1Gbit bandwidth (online phase only)

5.1 Testing Under Different Network Conditions

Table 2 shows how the end-to-end latency of the online phase changes according to the number of parties or under increased network latency. Note that bandwidth does not affect the latency of a single decryption, since throughout the protocol the parties only exchange a small number of ring elements. Conversely, the base online protocol requires three sequential public openings, and in this implementation, a couple more rounds for the SPD$\mathbb{Z}_{2^k}$ MAC check, which puts a lower-bound on the achievable latency.

[4]https://github.com/htsstfhed/acm-ccs-2025

[5]for example in https://docs.zama.ai/tfhe-rs.

	4 Parties	8 Parties	16 Parties
Ping Time 1 ms	8.483	10.086	24.230
Ping Time 10 ms	55.616	55.490	56.979

Table 2: Benchmark results for online protocol decryption time (in ms) of a single ciphertext

Achieving very high throughput is where our protocol shines. Since in the online phase, we only communicate a constant number of ring elements, and we perform very little computation (unlike previous works), then we can scale our throughput to thousands and even tens of thousands of decryptions per second on a single server, regardless of network latency. This is captured in Table 3, measuring throughput under various conditions.

5.2 Preprocessing

To empirically benchmark the procedures constructing our Sign and ModLTZ gates, we implemented these protocols using MP-SPDZ [31] and benchmarked them under SPD$\mathbb{Z}_{2^k}$ setting. To provide a good estimate about how long our own protocols take, we neutralized the time it takes to pre-process random bits and triplets, as there are many different ways to do so (e.g., [22, 27, 42]). All tests were run with the default MP-SPDZ value of two parties, 1ms ping time and 1Gbps bandwidth.

Table 4 shows how many gates per second we can produce, depending on the value b (recall that each gate produces 2^b entries). Both types of gates take roughly the same time to produce, as the bulk of the work is in PrepSubsetProds. In Table 5 we normalize the results based on our online phase parameters, which shows that it only takes approximately **1.35 seconds** to produce enough material per 1,000 decryptions.

6 RELATED WORK

Threshold FHE is a well studied problem, as it is a cornerstone for constructing constant-round MPC (e.g., [2, 37]) and threshold cryptography more generally (e.g., [6, 13, 30]). And yet, all prior simulation-secure works utilized some flavor of noise-flooding to design a secure decryption protocol. The first proposed solution in the literature provided an actively secure protocol (with abort) for a small number of parties (due to the use of replicated secret sharing) [5]. This was later expanded by [2] to full security and any number of parties. Both works utilized somewhat expensive zero-knowledge proofs for active security, which was sufficient for low-depth circuits and non-realtime applications such as SPDZ pre-processing [25], but not in general. In [17], the authors achieve a *robust* protocol with much better efficiency for active security by assuming a super-honest majority threshold ($t < n/3$), and utilizing well-known error-correction techniques.

In [6] and more recently in [13], the authors construct a generic universal thresholdizer that can convert any cryptosystem into a threshold variant. Like all other schemes before them, they require a super-polynomial gap between the noise and the message for noise flooding, making them too in-efficient in practice. In addition, these works propose new variants of linear secret-sharing that are suboptimal in terms of share size, leading to additional computational and communication costs in practice (and a larger dependence on the total number of parties), and also potentially raise security concerns [16].

In the last couple of years, likely as a result of recent advances in practical FHE and its applications [1, 36, 44, 46, 49], we have seen a significant increase in works focused on efficient threshold FHE decryption. However, all of these works weaken the model or security assumptions in at least one way (game-based security, assuming additional trusted parties, limiting the number of decryptions, or adding new non-standard assumptions). In [7, 18, 21], the authors circumvent the need for selecting the masking (smudging) noise from a large range, but as a result they are only able to prove a certain form of game-based security, making it hard to reason about using their scheme in a larger protocol. These schemes suffer from other limitations and potential vulnerabilities, as discussed in [39]. Alternatively, Micciancio and Suhl [33] were able to construct a very efficient semi-honest and simulation secure threshold LWE decryption scheme without noise-flooding. However, their scheme is not proven secure when applying homomorphic operations on ciphertexts. Additionally, they also rely on a new security assumption they describe as *Known-Norm LWE*.

Several other works tweaked the standard notion of threshold FHE decryption in pursuit of efficiency by designing very specialized models that are not standard and therefore not comparable to ours, e.g.,[47] proposes client-aided threshold FHE, and [39] assumes an incorruptible server assisting the parties.

We consider the recent work of [20] to represent the current state-of-the-art in terms of threshold FHE decryption. They also use the technique from [17] to present a robust protocol (assuming $t < n/3$). While they greatly improve the blowup in LWE parameters needed for noise-flooding, their results in both latency, and in particular around throughput, are orders of magnitude slower than ours (see Section 5). Another limitation is that their protocol is limited to an honest-majority adversary, while ours can support any type of adversary depending on the underlying $\mathcal{F}_{\text{ABB}}$ realization.

Moreover, in contrast to this long line of works, ours is the only protocol to construct an efficient threshold FHE decryption scheme without noise flooding, which is proven to be UC-secure. We support any adversary depending on the underlying $\mathcal{F}_{\text{ABB}}$ realization, including a dishonest-majority adversary via SPD$\mathbb{Z}_{2^k}$, and an honest-majority adversary via standard Shamir secret-sharing techniques over the Galois ring [26, 45]. Assuming $t < n/3$, our protocol is also robust via the same error-correcting techniques used in other works [17, 20].

REFERENCES

[1] A. Al Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, et al. OpenFHE: Open-source fully homomorphic encryption library. In *proceedings of the 10th workshop on encrypted computing & applied homomorphic cryptography*, pages 53–63, 2022.

[2] G. Asharov, A. Jain, A. López-Alt, E. Tromer, V. Vaikuntanathan, and D. Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In *Advances in Cryptology–EUROCRYPT 2012: 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings 31*, pages 483–501. Springer, 2012.

[3] D. Beaver. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology—CRYPTO'91: Proceedings 11*, pages 420–432. Springer, 1992.

[4] R. Bendlin and I. Damgård. Threshold decryption and zero-knowledge proofs for lattice-based cryptosystems. In *Theory of Cryptography Conference*, pages

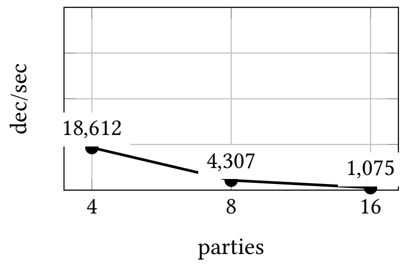
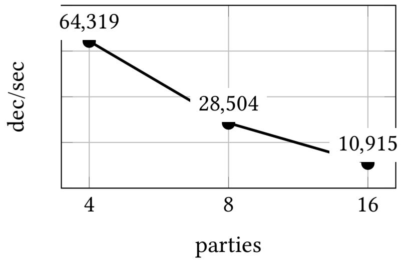
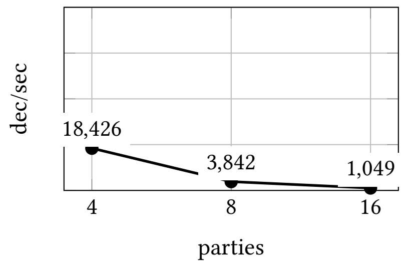
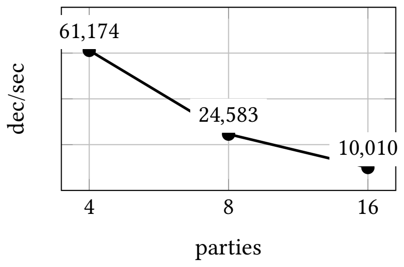
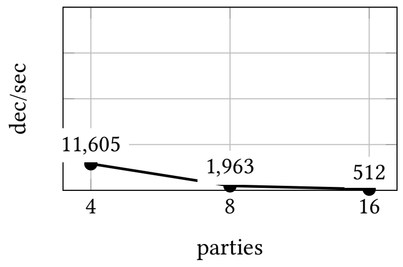
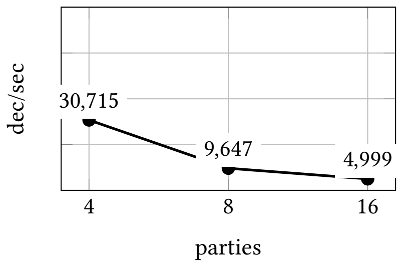
	100 MBit/s	1 Gbit/s
1 ms		
10 ms		
100 ms		

Table 3: Benchmark results of online protocol decryption time under varying network conditions, measured across different number of parties (4, 8, 16), bandwidths (100 Mbit/s, 1 Gbit/s), and latencies (1 ms, 10 ms, 100 ms).

bits (b)	Gates/sec	Data sent per gate (KB)
4	94 004	0.35
5	44 770	0.83
6	13 428	1.82
7	10 966	3.84
8	6350	7.90
9	3400	16.06

Table 4: Gate preprocessing costs for different values of b.

Gate	b	Gates/ dec	Time per 1K Decs (s)	Data per 1K Decs (MB)
Sign	8	8	1.06	63.23
ModLTZ	9	1	0.29	16.06
Total		**9**	**1.35**	**79.29**

Table 5: Preprocessing costs per 1K decryptions.

201–218. Springer, 2010.

[5] R. Bendlin and I. Damgård. Threshold decryption and zero-knowledge proofs for lattice-based cryptosystems. In *Theory of Cryptography Conference*, pages 201–218. Springer, 2010.

[6] D. Boneh, R. Gennaro, S. Goldfeder, A. Jain, S. Kim, P. M. Rasmussen, and A. Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. In *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part I 38*, pages 565–596. Springer, 2018.

[7] K. Boudgoust and P. Scholl. Simple threshold (fully homomorphic) encryption from LWE with polynomial modulus. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 371–404. Springer, 2023.

[8] E. Boyle, N. Chandran, N. Gilboa, D. Gupta, Y. Ishai, N. Kumar, and M. Rathee. Function secret sharing for mixed-mode and fixed-point secure computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 871–900. Springer, 2021.

[9] E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 337–367. Springer, 2015.

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

[11] R. Canetti, R. Gennaro, S. Goldfeder, N. Makriyannis, and U. Peled. Uc non-interactive, proactive, threshold ECDSA with identifiable aborts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1769–1787, 2020.

[12] O. Catrina and S. De Hoogh. Improved primitives for secure multiparty integer computation. In *Security and Cryptography for Networks: 7th International Conference, SCN 2010, Amalfi, Italy, September 13-15, 2010. Proceedings 7*, pages

182–199. Springer, 2010.

[13] J. H. Cheon, W. Cho, and J. Kim. Improved universal thresholdizer from iterative shamir secret sharing. *Cryptology ePrint Archive*, 2023.

[14] K. Chida, K. Hamada, D. Ikarashi, R. Kikuchi, D. Genkin, Y. Lindell, and A. Nof. Fast large-scale honest-majority MPC for malicious adversaries. *Journal of Cryptology*, 36(3):15, 2023.

[15] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. TFHE: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.

[16] W. Cho, J. Kim, and C. Lee. (in)security of threshold fully homomorphic encryption based on shamir secret sharing. *Cryptology ePrint Archive*, 2024.

[17] A. Choudhury, J. Loftus, E. Orsini, A. Patra, and N. P. Smart. Between a rock and a hard place: Interpolating between MPC and FHE. In *International conference on the theory and application of cryptology and information security*, pages 221–240. Springer, 2013.

[18] S. Chowdhury, S. Sinha, A. Singh, S. Mishra, C. Chaudhary, S. Patranabis, P. Mukherjee, A. Chatterjee, and D. Mukhopadhyay. Efficient threshold FHE with application to real-time systems. *Cryptology ePrint Archive*, 2022.

[19] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing. Spd$\mathbb{Z}_{2^k}$: efficient mpc mod 2^k for dishonest majority. In *Annual International Cryptology Conference*, pages 769–798. Springer, 2018.

[20] M. Dahl, D. Demmler, S. El Kazdadi, A. Meyre, J.-B. Orfila, D. Rotaru, N. P. Smart, S. Tap, and M. Walter. Noah's Ark: Efficient threshold-FHE using noise flooding. In *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 35–46, 2023.

[21] X. Dai, W. Wu, and Y. Feng. Key lifting: Multi-key fully homomorphic encryption in plain model without noise flooding. *Cryptology ePrint Archive*, 2022.

[22] I. Damgård, D. Escudero, T. Frederiksen, M. Keller, P. Scholl, and N. Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1102–1120. IEEE, 2019.

[23] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography Conference*, pages 285–304. Springer, 2006.

[24] I. Damgård and J. B. Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In *Annual international cryptology conference*, pages 247–264. Springer, 2003.

[25] I. Damgård, V. Pastro, N. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual Cryptology Conference*, pages 643–662. Springer, 2012.

[26] D. Escudero. Multiparty computation over $\mathbb{Z}/2^k\mathbb{Z}$. *University of Aarhus*, 2021.

[27] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part II 40*, pages 823–852. Springer, 2020.

[28] D. Escudero, C. Xing, and C. Yuan. More efficient dishonest majority secure computation over $\mathbb{Z}_{2^k}$ via Galois rings. In *Annual International Cryptology Conference*, pages 383–412. Springer, 2022.

[29] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[30] K. D. Gür, J. Katz, and T. Silde. Two-round threshold lattice signatures from threshold homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2023:1318, 2023.

[31] M. Keller. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 1575–1590, 2020.

[32] H. Lipmaa and T. Toft. Secure equality and greater-than tests with sublinear online complexity. In *Automata, Languages, and Programming: 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II 40*, pages 645–656. Springer, 2013.

[33] D. Micciancio and A. Suhl. Simulation-secure threshold PKE from LWE with polynomial modulus. *Cryptology ePrint Archive*, 2023.

[34] P. Mohassel and Y. Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*, pages 19–38. IEEE, 2017.

[35] C. Mouchet, J. Troncoso-Pastoriza, J.-P. Bossuat, and J.-P. Hubaux. Multiparty homomorphic encryption from ring-learning-with-errors. *Proceedings on Privacy Enhancing Technologies*, 2021(4):291–311, 2021.

[36] C. V. Mouchet, J.-P. Bossuat, J. R. Troncoso-Pastoriza, and J.-P. Hubaux. Lattigo: A multiparty homomorphic encryption library in go. In *Proceedings of the 8th Workshop on Encrypted Computing and Applied Homomorphic Cryptography*, pages 64–70, 2020.

[37] P. Mukherjee and D. Wichs. Two round multiparty computation via multi-key FHE. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*, pages 735–763. Springer, 2016.

[38] T. Nishide and K. Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *Public Key Cryptography–PKC 2007: 10th International Conference on Practice and Theory in Public-Key Cryptography Beijing, China, April 16-20, 2007. Proceedings 10*, pages 343–360. Springer, 2007.

[39] A. Passelègue and D. Stehlé. Low communication threshold fully homomorphic encryption. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 297–329. Springer, 2025.

[40] C. Peikert. A decade of lattice cryptography. *Foundations and Trends in Theoretical Computer Science*, 10(4):283–424, 2016.

[41] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6):1–40, 2009. Preliminary version in STOC 2005.

[42] D. Rotaru and T. Wood. Marbled circuits: Mixing arithmetic and boolean circuits with active security. In *International Conference on Cryptology in India*, pages 227–249. Springer, 2019.

[43] T. Ryffel, P. Tholoniat, D. Pointcheval, and F. Bach. Ariann: Low-interaction privacy-preserving deep learning via function secret sharing. *arXiv preprint arXiv:2006.04593*, 2020.

[44] S. Sav, A. Pyrgelis, J. R. Troncoso-Pastoriza, D. Froelicher, J.-P. Bossuat, J. S. Sousa, and J.-P. Hubaux. POSEIDON: Privacy-preserving federated neural network learning. *arXiv preprint arXiv:2009.00349*, 2020.

[45] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[46] R. Solomon, R. Weber, and G. Almashaqbeh. smartFHE: Privacy-preserving smart contracts from fully homomorphic encryption. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pages 309–331. IEEE, 2023.

[47] Y. Sugizaki, H. Tsuchida, T. Hayashi, K. Nuida, A. Nakashima, T. Isshiki, and K. Mori. Threshold fully homomorphic encryption over the torus. In *European Symposium on Research in Computer Security*, pages 45–65. Springer, 2023.

[48] S. Wagh, D. Gupta, and N. Chandran. SecureNN: 3-party secure computation for neural network training. *Proceedings on Privacy Enhancing Technologies*, 2019.

[49] G. Zyskind, Y. Erez, T. Langer, I. Grossman, and L. Bondarevsky. FHE-Rollups: Scaling confidential smart contracts on ethereum and beyond.

[50] G. Zyskind, T. South, and A. Pentland. Don't forget private retrieval: distributed private similarity search for large language models. *arXiv preprint arXiv:2311.12955*, 2023.

[51] G. Zyskind, A. Yanai, and A. Pentland. High-throughput three-party DPFs with applications to ORAM and digital currencies. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 4152–4166, 2024.

A SECURITY OF Π_{Decrypt}

To establish the security of our protocol, we prove Theorem 1 in the UC framework [11]. We construct a simulator $\mathcal{S}$ that runs the adversary $\mathcal{A}$ internally and interacts with the environment $\mathcal{Z}$ and $\mathcal{F}_{\text{Decrypt}}$ in the ideal-world. We show that $\mathcal{S}$ produces a simulated view (inputs, protocol messages and outputs) of the adversary and the output of the honest parties that is indistinguishable from those produced in the real execution of the protocol.

Proof. We assume that $\mathcal{S}$ has access to a simulator $\mathcal{S}_{\text{KeyGen}}$ proving the security of the $\mathcal{F}_{\text{ABB}}$-hybrid protocol for KeyGen used by the parties in the real protocol. Additionally, for brevity purposes, we assume that $\mathcal{S}$ forwards all commands (issued by $\mathcal{Z}$ to a corrupted party) to $\mathcal{F}_{\text{Decrypt}}$, without modification. In addition, $\mathcal{S}$ simulates the execution flow of the protocol and simulates sending the proper LinComb, Mult, or RandBit responses from $\mathcal{F}_{\text{ABB}}$ to the corrupted parties . If it receives messages from the corrupted parties to $\mathcal{F}_{\text{ABB}}$ other than opening, it simply ignores them. The simulator also consistently simulates any shell-dependent edge cases (e.g., *aborts*).

(1) In the *Initizalize* phase, on receiving command $(Init, \text{sid})$ issued by $\mathcal{Z}$ to a corrupted party P_i, the simulator runs $\mathcal{S}_{\text{KeyGen}}$ to simulate that part of the protocol. It ignores future calls to this command.

(2) In the *Decrypt* phase, on receiving command $(Decrypt, \text{sid}, \mathbf{c})$:

- On the first call to $\mathcal{F}_{\text{ABB}}$.Open to open $[z']_l$ received from the corrupted parties, sample a uniform value $\hat{z'} \in [L]$ and simulate sending it to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABB}}$.
- On the second call to $\mathcal{F}_{\text{ABB}}$.Open to open $[y']_{d+1}$ (occurs when applying the ModLTZ gate in Procedure 3) received from the corrupted parties, sample a uniform value $\hat{y'} \in [2^{d+1}]$ and simulate sending it to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABB}}$.
- On the final call to $\mathcal{F}_{\text{ABB}}$.Open to open $[\mu']_q$ (the shifted decrypted plaintext) received from the corrupted parties, receive the rounded inner product $\hat{\mu} = \lfloor \langle \mathbf{c}, \mathbf{s} \rangle \rceil_p$ from the functionality $\mathcal{F}_{\text{Decrypt}}$.Decrypt, compute $\hat{\mu'} = \hat{\mu} \cdot L$ and simulate sending it to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABB}}$. Output $\hat{\mu}$.

The indistinguishability between an execution of the real protocol Π_{Decrypt} and the simulation is shown via a sequence of hybrid games starting from the real world.

(1) **H0**: This is Π_{Decrypt}.
(2) **H1**: We replace the opening of $[z']_l = [z]_l + [r]_l$ with a value $[\hat{z'}]_l$. Since $[r]_l$ is uniform random value in $[L]$, it perfectly masks $[z]_l$ in the real execution. Thus, z' in $[L]$ is indistinguishable from the simulated $[\hat{z'}]_l$, and $H1 \sim H0$.
(3) **H2**: We replace the opening of $[y']_{d+1}$ with a value $[\hat{y'}]_{d+1}$. By a similar argument as before, y' is a uniform random value in $[2^{d+1}]$, just like $\hat{y'}_{d+1}$. Therefore, $H2 \sim H1$.
(4) **H3**: We replace the final opening of μ' of the real protocol with the rounded inner product $\lfloor \langle \mathbf{c}, \mathbf{s} \rangle \rceil_p \cdot L$.
As shown in Section 3.2, Π_{Decrypt} first computes $[z]_k = [\langle \mathbf{c}, \mathbf{s} \rangle]_k$ and then computes $\text{Mod}_{l,k}$ on $[z]_k$ (with the preprocessed gates) to obtain $[e]_k = [z \bmod 2^l]_k$, which is proven correct by Lemma 2. The parties then obtain $\mu' \in \mathbb{Z}_q$, which is equivalent to $\hat{\mu'} = \hat{\mu} \cdot L = [\lfloor \langle \mathbf{c}, \mathbf{s} \rangle]\rceil_p \cdot L$. Therefore, $H3 \sim H2$.

After the last opening, Π_{Decrypt} divides μ' by the constant L, receiving $\mu = \lfloor \langle \mathbf{c}, \mathbf{s} \rangle \rceil_p$, which is perfectly indistinguishable from the output of $\mathcal{F}_{\text{Decrypt}}$.*Decrypt* on $\mathbf{c}$. □