Practical Covertly Secure MPC for Dishonest Majority – or: Breaking the SPDZ Limits

Ivan Damgård[1], Marcel Keller[2], Enrique Larraia[2], Valerio Pastro[1], Peter Scholl[2], and Nigel P. Smart[2]

[1] Department of Computer Science, Aarhus University
[2] Department of Computer Science, University of Bristol

Abstract. SPDZ (pronounced "Speedz") is the nickname of the MPC protocol of Damgård et al. from Crypto 2012. SPDZ provided various efficiency innovations on both the theoretical and practical sides compared to previous work in the preprocessing model. In this paper we both resolve a number of open problems with SPDZ; and present several theoretical and practical improvements to the protocol.

In detail, we start by designing and implementing a covertly secure key generation protocol for obtaining a BGV public key and a shared associated secret key. In prior work this was assumed to be provided by a given setup functionality. Protocols for generating such shared BGV secret keys are likely to be of wider applicability than to the SPDZ protocol alone.

We then construct both a covertly and actively secure preprocessing phase, both of which compare favourably with previous work in terms of efficiency and provable security.

We also build a new online phase, which solves a major problem of the SPDZ protocol: namely prior to this work preprocessed data could be used for only one function evaluation and then had to be recomputed from scratch for the next evaluation, while our online phase can support reactive functionalities. This improvement comes mainly from the fact that our construction does not require players to reveal the MAC keys to check correctness of MAC'd values.

Since our focus is also on practical instantiations, our implementation offloads as much computation as possible into the preprocessing phase, thus resulting in a faster online phase. Moreover, a better analysis of the parameters of the underlying cryptoscheme and a more specific choice of the field where computation is performed allow us to obtain a better optimized implementation. Improvements are also due to the fact that our construction is in the random oracle model, and the practical implementation is multi-threaded.

This article is based on an earlier article: ESORICS 2013, pp 1–18, Springer LNCS 8134, 2013, http://dx.doi.org/10.1007/978-3-642-40203-6_1.

1 Introduction

For many decades multi-party computation (MPC) had been a predominantly theoretic endeavour in cryptography, but in recent years interest has arisen on the practical side. This has resulted in various implementation improvements and such protocols are becoming more applicable to practical situations. A key part in this transformation from theory to practice is in adapting theoretical protocols and applying implementation techniques so as to significantly improve performance, whilst not sacrificing the level of security required by real world applications. This paper follows this modern, more practical, trend.

Early applied work on MPC focused on the case of protocols secure against passive adversaries, both in the case of two-party protocols based on Yao circuits [19] and that of many-party protocols, based on secret sharing techniques [5, 10, 25]. Only in recent years work has shifted to achieve active security [17, 18, 24], which appears to come at vastly increased cost when dealing with more than two players. On the other hand, in the real applications active security may be more stringent than one would actually require. In [2, 3] Aumann and Lindell introduced the notion of covert security; in this security model an adversary who deviates from the protocol is detected with high (but not necessarily overwhelming) probability, say 90%, which still translates into an incentive on the adversary to behave in an honest manner. In contrast active security achieves the same effect, but the adversary can only succeed with cheating with negligible probability. There is a strong case to be made, see [2, 3], that covert security is a "good enough" security level for practical application; thus in this work we focus on covert security, but we also provide solutions with active security.

As our starting point we take the protocol of [14] (dubbed SPDZ, and pronounced Speedz). In [14] this protocol is secure against active static adversaries in the standard model, is actively secure, and tolerates corruption of $n - 1$ of the n parties. The SPDZ protocol follows the preprocessing model: in an offline phase some shared randomness is generated, but neither the function to be computed nor the inputs need be known; in an online phase the actual secure computation is performed. One of the main advantages of the SPDZ protocol is that the performance of the online phase scales linearly with the number of players, and the basic operations are almost as cheap as those used in the passively secure protocols based on Shamir secret sharing. Thus, it offers the possibility of being both more flexible and secure than Shamir based protocols, while still maintaining low computational cost.

In [12] the authors present an implementation report on an adaption of the SPDZ protocol in the random oracle model, and show performance figures for both the offline and online phases for both an actively secure variant and a covertly secure variant. The implementation is over a finite field of characteristic two, since the focus is on providing a benchmark for evaluation of the AES circuit (a common benchmark application in MPC [24, 11]).

Our Contributions: In this work we present a number of contributions which extend even further the ability the SPDZ protocol to deal with the type of application one is likely to see in practice. All our theorems are proved in the UC model, and in most cases, the protocols make use of some predefined ideal functionalities. We give protocols implementing most of these functionalities, the only exception being the functionality that provides access to a random oracle. This is implemented using a hash functions, and so the actual protocol is only secure in the Random Oracle Model. We back up these improvements with an implementation which we report on.

Our contributions come in two flavours. In the first flavour we present a number of improvements and extensions to the basic underlying SPDZ protocol. These protocol improvements are supported with associated security models and proofs. Our second flavour of improvements are at the implementation layer, and they bring in standard techniques from applied cryptography to bear onto MPC.

In more detail our protocol enhancements, in what are the descending order of importance, are as follows:

1. In the online phase of the original SPDZ protocol the parties are required to reveal their shares of a global MAC key in order to verify that the computation has been performed correctly. This is a major problem in practical applications since it means that secret-shared data we did not reveal cannot be re-used in later applications. Our protocol adopts a method to accomplish the same task, without needing to open the underlying MAC key. This means we can now go on computing on any secret-shared data we have, so we can support general reactive computation rather than just secure function evaluation. A further advantage of this technique is that some of the verification we need (the so-called "sacrificing" step) can be moved into the offline phase, providing additional performance improvements in the online phase.

2. In the original SPDZ protocol [12, 14] the authors assume a "magic" key generation phase for the production of the distributed Somewhat Homomorphic Encryption (SHE) scheme public/private keys required by the offline phase. The authors claim this can be accomplished using standard generic MPC techniques, which are of course expensive. In this work we present a key generation protocol for the BGV [6] SHE scheme, which is secure against covert adversaries. In addition we generate a "full" BGV key which supports the modulus switching and key switching used in [16]. This new sub-protocol may be of independent interest in other applications which require distributed decryption in an SHE/FHE scheme.
3. In [12] the modification to covert security was essentially ad-hoc, and resulted in a very weak form of covert security. In addition no security proofs or model were given to justify the claimed security. In this work we present a completely different approach to achieving covert security, we provide an extensive security model and provide full proofs for the modified offline phase (and the key generation protocol mentioned above).
4. We introduce a new approach to obtain full active security in the offline phase. In [14] active security was obtained via the use of specially designed ZKPoKs. In this work we present a different technique, based on a method used in [21]. This method has running time similar to the ZKPoK approach utilized in [14], but it allows us to give much stronger guarantees on the ciphertexts produced by corrupt players: the gap between the size of "noise" honest players put into ciphertexts and what we can force corrupt players to use was exponential in the security parameter in [14], and is essentially linear in our solution. This allows us to choose smaller parameters for the underlying cryptosystem and so makes other parts of the protocol more efficient.

It is important to understand that by combining these contributions in different ways, we can obtain two different general MPC protocols: First, since our new online phase still has full active security, it can be combined with our new approach to active security in the offline phase. This results in a protocol that is "syntactically similar" to the one from [14]: it has full active security assuming access to a functionality for key generation. However, it has enhanced functionality and performance, compared to [14], in that it can securely compute reactive functionalities. Second, we can combine our covertly secure protocols for key generation and the offline phase with the online phase to get a protocol that has covert security throughout and does not assume that key generation is given for free.

Our covert solutions all make use of the same technique to move from passive to covert security, while avoiding the computational cost of performing zero-knowledge proofs. In [12] covert security is obtained by only checking a fraction of the resulting proofs, which results in a weak notion of covert security (the probability of a cheater being detected cannot be made too large). In this work we adopt a different approach, akin to the cut-and-choose paradigm. We require parties to commit to random seeds for a number of runs of a given sub-protocol, then all the runs are executed in parallel, finally all bar one of the runs are "opened" by the players revealing their random seeds. If all opened runs are shown to have been performed correctly then the players assume that the single un-opened run is also correctly executed.

Note that since these checks take place in the offline phase where the inputs are not yet available, we obtain the strongest flavour of covert security defined in [2], where the adversary learns nothing new if he decides to try to cheat and is caught.

A pleasing side-effect of the replacement of zero-knowledge proofs with our custom mechanism to obtain covert security is that the offline phase can be run in much smaller "batches". In [12, 14] the need to amortize the cost of the expensive zero-knowledge proofs meant that the players on each iteration of the offline protocol executed a large computation, which produced a large number of multiplication triples [4] (in the millions). With our new technique we no longer need to amortize executions as much, and so short runs of the offline phase can be executed if so desired; producing only a few thousand triples per run.

Our second flavour of improvements at the implementation layer are more mundane; being mainly of an implementation nature.

1. We focus on the more practical application scenario of developing MPC where the base arithmetic domain is a finite field of characteristic $p > 2$. The reader should think $p \approx 2^{32}, 2^{64}, 2^{128}$ and the type of operations envisaged in [8, 9] etc. For such applications we can offload a lot of computation into the SPDZ offline phase, and we present the necessary modifications to do so.
2. Parameters for the underlying BGV scheme are chosen using the analysis used in [16] rather than the approach used in [14]. In addition we pick specific parameters which enable us to optimize for our application to SPDZ with the choices of p above.

3. We assume the random oracle model throughout, this allows us to simplify a number of the sub-procedures in [14]; especially, related to aspects of the protocol which require commitments.
4. The underlying modular arithmetic is implemented using Montgomery arithmetic [20], this is contrasted to earlier work which used standard libraries, such as NTL, to provide such operations.
5. The removal of the need to use libraries such as NTL means the entire protocol can be implemented in a multi-threaded manner; thus it can make use of the multiple cores on modern microprocessors.

This extended abstract presents the main ideas behind our improvements and details of our implementation. For a full description including details of the associated sub-procedures, security models and associated full security proofs please see the full version of this paper at [13].

2 SPDZ Overview

We now present the main components of the SPDZ protocol; in this section unless otherwise specified we are simply recapping on prior work. Throughout the paper we assume the computation to be performed by n players over a fixed finite field $\mathbb{F}_p$ of characteristic p. The high level idea of the online phase is to compute a function represented as a circuit, where privacy is obtained by additively secret sharing the inputs and outputs of each gate, and correctness is guaranteed by adding additive secret sharings of MACs on the inputs and outputs of each gate. In more detail, each player P_i has a uniform share $\alpha_i \in \mathbb{F}_p$ of a secret value $\alpha = \alpha_1 + \cdots + \alpha_n$, thought of as a fixed MAC key. We say that a data item $a \in \mathbb{F}_p$ is $\langle\cdot\rangle$-shared if P_i holds a tuple $(a_i, \gamma(a)_i)$, where a_i is an additive secret sharing of a, i.e. $a = a_1 + \cdots + a_n$, and $\gamma(a)_i$ is an additive secret sharing of $\gamma(a) := \alpha \cdot a$, i.e. $\gamma(a) = \gamma(a)_1 + \cdots + \gamma(a)_n$.

For the readers familiar with [14], this is a simpler MAC definition. In particular we have dropped δ_a from the MAC definition; this value was only used to add or subtract public data to or from shares. In our case δ_a becomes superfluous, since there is a straightforward way of computing a MAC of a public value a by defining $\gamma(a)_i \leftarrow a \cdot \alpha_i$.

During the protocol various values which are $\langle\cdot\rangle$-shared are "partially opened", i.e. the associated values a_i are revealed, but not the associated shares of the MAC. Note that linear operations (addition and scalar multiplication) can be performed on the $\langle\cdot\rangle$-sharings with no interaction required. Computing multiplications, however, is not straightforward, as we describe below.

The goal of the offline phase is to produce a set of "multiplication triples", which allow players to compute products. These are a list of sets of three $\langle\cdot\rangle$-sharings $\{\langle a\rangle, \langle b\rangle, \langle c\rangle\}$ such that $c = a \cdot b$. In this paper we extend the offline phase to also produce "square pairs" i.e. a list of pairs of $\langle\cdot\rangle$-sharings $\{\langle a\rangle, \langle b\rangle\}$ such that $b = a^2$, and "shared bits" i.e. a list of single shares $\langle a\rangle$ such that $a \in \{0, 1\}$.

In the online phase these lists are consumed as MPC operations are performed. In particular to multiply two $\langle\cdot\rangle$-sharings $\langle x\rangle$ and $\langle y\rangle$ we take a multiplication triple $\{\langle a\rangle, \langle b\rangle, \langle c\rangle\}$ and partially open $\langle x\rangle - \langle a\rangle$ to obtain ϵ and $\langle y\rangle - \langle b\rangle$ to obtain δ. The sharing of $z = x \cdot y$ is computed from $\langle z\rangle \leftarrow \langle c\rangle + \epsilon \cdot \langle b\rangle + \delta \cdot \langle a\rangle + \epsilon \cdot \delta$.

The reason for us introducing square pairs is that squaring a value can then be computed more efficiently as follows: To square the sharing $\langle x\rangle$ we take a square pair $\{\langle a\rangle, \langle b\rangle\}$ and partially open $\langle x\rangle - \langle a\rangle$ to obtain ϵ. We then compute the sharing of $z = x^2$ from $\langle z\rangle \leftarrow \langle b\rangle + 2 \cdot \epsilon \cdot \langle x\rangle - \epsilon^2$. Finally, the "shared bits" are useful in computing high level operation such as comparison, bit-decomposition, fixed and floating point operations as in [1, 8, 9].

The offline phase produces the triples in the following way. We make use of a Somewhat Homomorphic Encryption (SHE) scheme, which encrypts messages in $\mathbb{F}_p$, supports distributed decryption, and allows computation of circuits of multiplicative depth one on encrypted data. To generate a multiplication triple each player P_i generates encryptions of random values a_i and b_i (their shares of a and b). Using the multiplicative property of the SHE scheme an encryption of $c = (a_1 + \cdots + a_n) \cdot (b_1 + \cdots + b_n)$ is produced. The players then use the distributed decryption protocol to obtain sharings of c. The shares of the MACs on a, b and c needed to complete the $\langle\cdot\rangle$-sharing are produced in much the same manner. Similar operations are performed to produce square pairs and shared bits. Clearly the above (vague) outline needs to be fleshed out to ensure the required covert security level. Moreover, in practice we generate many triples/pairs/shared-bits at once using the SIMD nature of the BGV SHE scheme.

3 BGV

We now present an overview of the BGV scheme as required by our offline phase. This is only sketched, the reader is referred to [6, 15, 16] for more details; our goal is to present enough detail to explain the key generation protocol later.

3.1 Preliminaries

Underlying Algebra: We fix the ring $R_q = (\mathbb{Z}/q\mathbb{Z})[X]/\Phi_m(X)$ for some cyclotomic polynomial $\Phi_m(X)$, where m is an parameter to be determined later (see Appendix G). Note that q may not necessarily be prime. Let $R = \mathbb{Z}[X]/\Phi_m(X)$, and $\phi(m)$ denote the degree of R over $\mathbb{Z}$, i.e. Euler's ϕ function. The message space of our scheme will be R_p for a prime p of approximately 32, 64 or 128-bits in length, whilst ciphertexts will lie in either $R_{q_0}^2$ or $R_{q_1}^2$, for one of two moduli q_0 and q_1. We select $R = \mathbb{Z}[X]/(X^{m/2}+1)$ for m a power of two, and $p = 1 \pmod m$. By picking m and p this way we have that the message space R_p offers $m/2$-fold SIMD parallelism, i.e. $R_p \cong \mathbb{F}_p^{m/2}$. In addition this also implies that the ring constant c_m from [14, 16] is equal to one.

We wish to generate a public key for a leveled BGV scheme for which n players each hold a share, which is itself a "standard" BGV secret key. As we are working with circuits of multiplicative depth at most one we only need two levels in the moduli chain $q_0 = p_0$ and $q_1 = p_0 \cdot p_1$. The modulus p_1 will also play the role of P in [16] for the SwitchKey operation. The value p_1 must be chosen so that $p_1 \equiv 1 \pmod p$, with the value of p_0 set to ensure valid distributed decryption.

Random Values: Each player is assumed to have a secure entropy source. In practice we take this to be `/dev/urandom`, which is a non-blocking entropy source found on Unix like operating systems. This is not a "true" entropy source, being non-blocking, but provides a practical balance between entropy production and performance for our purposes. In what follows we model this source via a procedure $s \leftarrow \mathsf{Seed}()$, which generates a new seed from this source of entropy. Calling this function sets the players global variable cnt to zero. Then every time a player generates a new random value in a protocol this is constructed by calling $\mathsf{PRF}_s(\mathsf{cnt})$, for some pseudo-random function PRF, and then incrementing cnt. In practice we use AES under the key s with message cnt to implement PRF.

The point of this method for generating random values is that the said values can then be verified to have been generated honestly by revealing s in the future and recomputing all the randomness used by a player, and verifying his output is consistent with this value of s.

From the basic PRF we define the following "induced" pseudo-random number generators, which generate elements according to the following distributions but seeded by the seed s:

- $\mathcal{HWT}_s(h, n)$: This generates a vector of length n with elements chosen at random from $\{-1, 0, 1\}$ subject to the condition that the number of non-zero elements is equal to h.
- $\mathcal{ZO}_s(0.5, n)$: This generates a vector of length n with elements chosen from $\{-1, 0, 1\}$ such that the probability of coefficient is $p_{-1} = 1/4$, $p_0 = 1/2$ and $p_1 = 1/4$.
- $\mathcal{DG}_s(\sigma^2, n)$: This generates a vector of length n with elements chosen according to the discrete Gaussian distribution with variance σ^2.
- $\mathcal{RC}_s(0.5, \sigma^2, n)$: This generates a triple of elements (v, e_0, e_1) where v is sampled from $\mathcal{ZO}_s(0.5, n)$ and e_0 and e_1 are sampled from $\mathcal{DG}_s(\sigma^2, n)$.
- $\mathcal{U}_s(q, n)$: This generates a vector of length n with elements generated uniformly modulo q.

If any random values are used which **do not** depend on a seed then these should be assumed to be drawn using a secure entropy source (again in practice assumed to be `/dev/urandom`). If we pull from one of the above distributions where we do not care about the specific seed being used then we will drop the subscript s from the notation.

Broadcast: When broadcasting data we assume two different models. In the online phase during partial opening we utilize the method described in [14]; in that players send their data to a nominated player who then broadcasts the reconstructed value back to the remaining players. For other applications of broadcast we assume each party broadcasts their values to all other parties directly. In all instances players maintain a running hash of all values sent and received in a broadcast (with a suitable modification for the variant used for partial opening). At the end of a protocol run these

running hashes are compared in a pair-wise fashion. This final comparison ensures that in the case of at least two honest parties the adversary must have been consistent in what was sent to the honest parties.

Commitments: In Figure 2 we present an ideal functionality $\mathcal{F}_{\text{COMMIT}}$ for commitment which will be used in all of our protocols. Our protocols will be UC secure, this is possible despite the fact that we allow dishonest majority because we assume a random oracle is available; in particular we model a hash function $\mathcal{H}_1$ as a random oracle and define a commitment scheme to implement the functionality $\mathcal{F}_{\text{COMMIT}}$ as follows: The commit function $\mathsf{Commit}(m)$ generates a random value r and computes $c \leftarrow \mathcal{H}_1(m\|r)$. It returns the pair (c, o) where o is the opening information $m\|r$. When the commitment c is opened the committer outputs the value o and the receiver runs $\mathsf{Open}(c, o)$ which checks whether $c = \mathcal{H}_1(o)$ and if the check passes it returns m. See Appendix A for details.

3.2 Key Generation

The key generation algorithm generates a public/private key pair such that the public key is given by $\mathfrak{pk} = (a, b)$, where a is generated from $\mathcal{U}(q_1, \phi(m))$ (i.e. a is uniform in R_{q_1}), and $b = a \cdot \mathfrak{s} + p \cdot \epsilon$ where ϵ is a "small" error term, and $\mathfrak{s}$ is the secret key such that $\mathfrak{s} = \mathfrak{s}_1 + \cdots + \mathfrak{s}_n$, where player P_i holds the share $\mathfrak{s}_i$. Recall since m is a power of 2 we have $\phi(m) = m/2$.

The public key is also augmented to an extended public key $\mathfrak{epk}$ by addition of a "quasi-encryption" of the message $-p_1 \cdot \mathfrak{s}^2$, i.e. $\mathfrak{epk}$ contains a pair $\mathfrak{enc} = (b_{\mathfrak{s},\mathfrak{s}^2}, a_{\mathfrak{s},\mathfrak{s}^2})$ such that $b_{\mathfrak{s},\mathfrak{s}^2} = a_{\mathfrak{s},\mathfrak{s}^2} \cdot \mathfrak{s} + p \cdot \epsilon_{\mathfrak{s},\mathfrak{s}^2} - p_1 \cdot \mathfrak{s}^2$, where $a_{\mathfrak{s},\mathfrak{s}^2} \leftarrow \mathcal{U}(q_1, \phi(m))$ and $\epsilon_{\mathfrak{s},\mathfrak{s}^2}$ is a "small" error term. The precise distributions of all these values will be determined when we discuss the exact key generation protocol we use.

3.3 Encryption and Decryption

$\mathsf{Enc}_{\mathfrak{pk}}(\mathbf{m})$: To encrypt an element $m \in R_p$, using the modulus q_1, we choose one "small polynomial" (with $0, \pm 1$ coefficients) and two Gaussian polynomials (with variance σ^2), via $(v, e_0, e_1) \leftarrow \mathcal{RC}_s(0.5, \sigma^2, \phi(m))$. Then we set $c_0 = b \cdot v + p \cdot e_0 + m,\ c_1 = a \cdot v + p \cdot e_1$, and set the initial ciphertext as $\mathfrak{c}' = (c_0, c_1, 1)$.

$\mathsf{SwitchModulus}((c_0, c_1), \ell)$: The operation $\mathsf{SwitchModulus}(\mathfrak{c})$ takes the ciphertext $\mathfrak{c} = ((c_0, c_1), \ell)$ defined modulo q_ℓ and produces a ciphertext $\mathfrak{c}' = ((c_0', c_1'), \ell - 1)$ defined modulo $q_{\ell-1}$, such that $[c_0 - \mathfrak{s} \cdot c_1]_{q_\ell} \equiv [c_0' - \mathfrak{s} \cdot c_1']_{q_{\ell-1}} \pmod{p}$. This is done by setting $c_i' = \mathsf{Scale}(c_i, q_\ell, q_{\ell-1})$ where Scale is the function defined in [16]; note we need the more complex function of Appendix E of the full version of [16] if working in dCRT representation as we need to fix the scaling modulo p as opposed to modulo two which was done in the main body of [16]. As we are only working with two levels this function can only be called when $\ell = 1$.

$\mathsf{Dec}_{\mathfrak{s}}(\mathfrak{c})$: Note, that this operation is never actually performed, since no-one knows the shared secret key $\mathfrak{s}$, but presenting it will be instructive: Decryption of a ciphertext (c_0, c_1, ℓ) at level ℓ is performed by setting $m' = [c_0 - \mathfrak{s} \cdot c_1]_{q_\ell}$, then converting m' to coefficient representation and outputting $m' \bmod p$.

$\mathsf{DistDec}_{\mathfrak{s}_i}(\mathfrak{c})$: We actually decrypt using a simplification of the distributed decryption procedure described in [14], since our final ciphertexts consist of only two elements as opposed to three in [14]. For input ciphertext (c_0, c_1, ℓ), player P_1 computes $\mathbf{v}_1 = c_0 - \mathfrak{s}_i \cdot c_1$ and each other player P_i computes $\mathbf{v}_i = -\mathfrak{s}_i \cdot c_1$. Each party P_i then sets $\mathbf{t}_i = \mathbf{v}_i + p \cdot \mathbf{r}_i$ for some random element $\mathbf{r}_i \in R$ with infinity norm bounded by $2^{\mathsf{sec}} \cdot B/(n \cdot p)$, for some statistical security parameter sec, and the values $\mathbf{t}_i$ are broadcast; the precise value B being determined in Appendix G. Then the message is recovered as $\mathbf{t}_1 + \cdots + \mathbf{t}_n \pmod{p}$.

3.4 Operations on Encrypted Data

Homomorphic addition follows trivially from the methods of [6, 16]. So the main remaining task is to deal with multiplication. We first define a $\mathsf{SwitchKey}$ operation.

$\underline{\mathsf{SwitchKey}(d_0, d_1, d_2)}$: This procedure takes as input an extended ciphertext $\mathfrak{c} = (d_0, d_1, d_2)$ defined modulo q_1; this is a ciphertext which is decrypted via the equation

$$[d_0 - \mathfrak{s} \cdot d_1 - \mathfrak{s}^2 \cdot d_2]_{q_1}.$$

The SwitchKey operation also takes the key-switching data $\mathfrak{enc} = (b_{\mathfrak{s},\mathfrak{s}^2}, a_{\mathfrak{s},\mathfrak{s}^2})$ above and produces a standard two element ciphertext which encrypts the same message but modulo q_0.

- $c_0' \leftarrow p_1 \cdot d_0 + b_{\mathfrak{s},\mathfrak{s}^2} \cdot d_2 \pmod{q_1}$, $\quad c_1' \leftarrow p_1 \cdot d_1 + a_{\mathfrak{s},\mathfrak{s}^2} \cdot d_2 \pmod{q_1}$.
- $c_0'' \leftarrow \mathsf{Scale}(c_0', q_1, q_0)$, $\quad c_1'' \leftarrow \mathsf{Scale}(c_1', q_1, q_0)$.
- Output $((c_0'', c_1''), 0)$.

Notice we have the following equality modulo q_1:

$$\begin{aligned} c_0' - \mathfrak{s} \cdot c_1' &= (p_1 \cdot d_0) + d_2 \cdot b_{\mathfrak{s},\mathfrak{s}^2} - \mathfrak{s} \cdot \big((p \cdot d_1) - d_2 \cdot a_{\mathfrak{s},\mathfrak{s}^2}\big) \\ &= p_1 \cdot (d_0 - \mathfrak{s} \cdot d_1 - \mathfrak{s}^2 d_2) - p \cdot d_2 \cdot \epsilon_{\mathfrak{s},\mathfrak{s}^2}, \end{aligned}$$

The requirement on $p_1 \equiv 1 \pmod{p}$ is from the above equation as we want this to produce the same value as $d_0 - \mathfrak{s} \cdot d_1 - \mathfrak{s}^2 d_2 \mod q_1$ on reduction modulo p.

$\underline{\mathsf{Mult}(\mathfrak{c}, \mathfrak{c}')}$: We only need to execute multiplication on two ciphertexts at level one, thus $\mathfrak{c} = ((c_0, c_1), 1)$ and $\mathfrak{c}' = \underline{((c_0', c_1'), 1)}$. The output will be a ciphertext $\mathfrak{c}''$ at level zero, obtained via the following steps:

- $\mathfrak{c} \leftarrow \mathsf{SwitchModulus}(\mathfrak{c})$, $\quad \mathfrak{c}' \leftarrow \mathsf{SwitchModulus}(\mathfrak{c}')$.
- $(d_0, d_1, d_2) \leftarrow (c_0 \cdot c_0', c_1 \cdot c_0' + c_0 \cdot c_1', -c_1 \cdot c_1')$.
- $\mathfrak{c}'' \leftarrow \mathsf{SwitchKey}(d_0, d_1, d_2)$.

4 Protocols Associated to the SHE Scheme

In this section we present two sub-protocols associated with the SHE scheme; namely our distributed key generation and a protocol for proving that a committed ciphertext is well formed.

4.1 Distributed Key Generation Protocol For BGV

To make the paper easier to follow we present the precise protocols, ideal functionalities, simulators and security proofs in Appendix B. Here we present a high level overview.

As remarked in the introduction, the authors of [14] assumed a "magic" set up which produces not only a distributed sharing of the main BGV secret key, but also a distributed sharing of the square of the secret key. That was assumed to be done via some other unspecified MPC protocol. The effect of requiring a sharing of the square of the secret key was that they did not need to perform KeySwitching, but ciphertexts were 50% bigger than one would otherwise expect. Here we take a very different approach: we augment the public key with the keyswitching data from [16] and provide an explicit covertly secure key generation protocol.

Our protocol will be covertly secure in the sense that the probability that an adversary can deviate without being detected will be bounded by $1/c$, for a positive integer c. Our basic idea behind achieving covert security is as follows: Each player runs c instances of the basic protocol, each with different random seeds, then at the end of the main protocol all bar a random one basic protocol runs are opened, along with the respective random seeds. All parties then check that the opened runs were performed honestly and, if any party finds an inconsistency, the protocol aborts. If no problem is detected, the parties assume that the single unopened run is correct. Thus intuitively the adversary can cheat with probability at most $1/c$.

We start by discussing the generation of the main public key $\mathfrak{pk}_j$ in execution j where $j \in \{1, \ldots, c\}$. To start with the players generate a uniformly random value $a_j \in R_{q_1}$. They then each execute the standard BGV key generation

procedure, except that this is done with respect to the global element a_j. Player i chooses a low-weight secret key and then generates an LWE instance relative to that secret key. Following [16], we choose

$$\mathfrak{s}_{i,j} \leftarrow \mathcal{HWT}_s(h, \phi(m)) \text{ and } \epsilon_{i,j} \leftarrow \mathcal{DG}_s(\sigma^2, \phi(m)).$$

Then the player sets the secret key as $\mathfrak{s}_{i,j}$ and their "local" public key as $(a_j, b_{i,j})$ where $b_{i,j} = [a_j \cdot \mathfrak{s}_{i,j} + p \cdot \epsilon_{i,j}]_{q_1}$.

Note, by a hybrid argument, obtaining n ring-LWE instances for n different secret keys but the same value of a_j is secure assuming obtaining one ring-LWE instance is secure. In the LWE literature this is called "amortization". Also note in what follows that a key modulo q_1 can be also treated as a key modulo q_0 since q_0 divides q_1 and $\mathfrak{s}_{i,j}$ has coefficients in $\{-1, 0, 1\}$.

The global public and private key are then set to be $\mathfrak{pk}_j = (a_j, b_j)$ and $\mathfrak{s}_j = \mathfrak{s}_{1,j} + \cdots + \mathfrak{s}_{n,j}$, where $b_j = [b_{1,j} + \cdots + b_{n,j}]_{q_1}$. This is essentially another BGV key pair, since if we set $\epsilon_j = \epsilon_{1,j} + \cdots + \epsilon_{n,j}$ then we have

$$b_j = \sum_{i=1}^{n} (a_j \cdot \mathfrak{s}_{i,j} + p \cdot \epsilon_{i,j}) = a_j \cdot \mathfrak{s}_j + p \cdot \epsilon_j,$$

but generated with different distributions for $\mathfrak{s}_j$ and ϵ_j compared to the individual key pairs above.

We next augment the above basic key generation to enable the construction of the KeySwitching data. Given a public key $\mathfrak{pk}_j$ and a share of the secret key $\mathfrak{s}_{i,j}$ our method for producing the extended public key is to produce in turn (see Figure 3 for the details on how we create these elements in our protocol).

- $\mathfrak{enc}'_{i,j} \leftarrow \mathsf{Enc}_{\mathfrak{pk}_j}(-p_1 \cdot \mathfrak{s}_{i,j})$
- $\mathfrak{enc}'_j \leftarrow \mathfrak{enc}'_{1,j} + \cdots + \mathfrak{enc}'_{n,j}$.
- $\mathfrak{zero}_{i,j} \leftarrow \mathsf{Enc}_{\mathfrak{pk}_j}(\mathbf{0})$
- $\mathfrak{enc}_{i,j} \leftarrow (\mathfrak{s}_{i,j} \cdot \mathfrak{enc}'_j) + \mathfrak{zero}_{i,j} \in R^2_{q_1}$.
- $\mathfrak{enc}_j \leftarrow \mathfrak{enc}_{1,j} + \cdots + \mathfrak{enc}_{n,j}$.
- $\mathfrak{epk}_j \leftarrow (\mathfrak{pk}_j, \mathfrak{enc}_j)$.

Note, that $\mathfrak{enc}'_{i,j}$ is not a valid encryption of $-p_1 \cdot \mathfrak{s}_{i,j}$, since $-p_1 \cdot \mathfrak{s}_{i,j}$ does not lie in the message space of the encryption scheme. However, because of the dependence on the secret key shares here, we need to assume a form of circular security; the precise assumption needed is stated in Appendix B. The encryption of zero, $\mathfrak{zero}_{i,j}$, is added on by each player to re-randomize the ciphertext, preventing an adversary from recovering $\mathfrak{s}_{i,j}$ from $\mathfrak{enc}_{i,j}/\mathfrak{enc}'_j$. We call the resulting $\mathfrak{epk}_j$ the *extended public key*. In [16] the keyswitching data $\mathfrak{enc}_j$ is computed directly from $\mathfrak{s}_j^2$; however, we need to use the above round-about way since $\mathfrak{s}_j^2$ is not available to the parties.

Finally we open all bar one of the c executions and check they have been executed correctly. If all checks pass then the final extended public key $\mathfrak{epk}$ is output and the players keep hold of their associated secret key share $\mathfrak{s}_i$. See Figure 3 for full details of the protocol.

Theorem 1. *In the $\mathcal{F}_{\text{COMMIT}}$-hybrid model, the protocol Π_{KEYGEN} implements $\mathcal{F}_{\text{KEYGEN}}$ with computational security against any static adversary corrupting at most $n-1$ parties.*

Recall that $\mathcal{F}_{\text{COMMIT}}$ is a standard functionality for commitment. $\mathcal{F}_{\text{KEYGEN}}$ simply generates a key pair with a distribution matching what we sketched above, and then sends the values $a_i, b_i, \mathfrak{enc}'_i, \mathfrak{enc}_i$ for every i to all parties and shares of the secret key to the honest players. Like most functionalities in the following, it allows the adversary to try to cheat and will allow this with a certain probability $1/c$. This is how we model covert security.

The BGV cryptosystem resulting from $\mathcal{F}_{\text{KEYGEN}}$ is proven semantically secure by the following theorem.

Theorem 2. *If the functionality $\mathcal{F}_{\text{KEYGEN}}$ is used to produce a public key $\mathfrak{epk}$ and secret keys $\mathfrak{s}_i$ for $i = 0, \ldots, n-1$ then the resulting cryptosystem is semantically secure based on the hardness of $\mathsf{RLWE}_{q_1,\sigma^2,h}$ and the circular security assumption in Appendix B.*

4.2 EncCommit

We use a sub-protocol $\Pi_{\text{EncCommit}}$ to replace the Π_{ZKPoPK} protocol from [14]. In this section we consider a covertly secure variant rather than active security; this means that players controlled by a malicious adversary succeed in deviating from the protocol with a probability bounded by $1/c$. In our experiments we pick $c = 5$, 10 and 20. In Appendix F we present an actively secure variant of this protocol.

Our new sub-protocol assumes that players have agreed on the key material for the encryption scheme, i.e. $\Pi_{\text{EncCommit}}$ runs in the $\mathcal{F}_{\text{KeyGen}}$-hybrid model. The protocol ensures that a party outputs a validly created ciphertext containing an encryption of some pseudo-random message m, where the message m is drawn from a distribution satisfying condition cond. This is done by committing to seeds and using the cut-and-choose technique, similarly to the key generation protocol. The condition cond in our application could either be uniformly pseudo-randomly generated from R_p, or uniformly pseudo-randomly generated from $\mathbb{F}_p$ (i.e. a "diagonal" element in the SIMD representation).

The protocol $\Pi_{\text{EncCommit}}$ and ideal functionality it implements are presented in Appendix C, along with the proof of the following theorem.

Theorem 3. *In the $(\mathcal{F}_{\text{Commit}}, \mathcal{F}_{\text{KeyGen}})$-hybrid model, the protocol $\Pi_{\text{EncCommit}}$ implements $\mathcal{F}_{\text{SHE}}$ with computational security against any static adversary corrupting at most $n-1$ parties.*

$\mathcal{F}_{\text{SHE}}$ offers the same functionality as $\mathcal{F}_{\text{KeyGen}}$ but can in addition generate correctly formed ciphertexts where the plaintext satisfies a condition cond as explained above, and where the plaintext is known to a particular player (even if he is corrupt). Of course, if we use the actively secure version of $\Pi_{\text{EncCommit}}$ from Appendix F, we would get a version of $\mathcal{F}_{\text{SHE}}$ where the adversary is not allowed to attempt cheating.

5 The Offline Phase

The offline phase produces pre-processed data for the online phase (where the secure computation is performed). To ensure security against active adversaries the MAC values of any partially opened value need to be verified. We suggest a new method for this that overcomes some limitations of the corresponding method from [14]. Since it will be used both in the offline and the online phase, we explain it here, before discussing the offline phase.

5.1 MAC Checking

We assume some value a has been $\langle\cdot\rangle$-shared and partially opened, which means that players have revealed shares of the a but not of the associated MAC value γ, this is still additively shared. Since there is no guarantee that the a are correct, we need to check it holds that $\gamma = \alpha a$ where α is the global MAC key that is also additively shared. In [14], this was done by having players commit to the shares of the MAC. then open α and check everything in the clear. But this means that other shared values become useless because the MAC key is now public, and the adversary could manipulate them as he desires.

So we want to avoid opening α, and observe that since a is public, the value $\gamma - \alpha a$ is a linear function of shared values γ, α, so players can compute shares in this value locally and we can then check if it is 0 without revealing information on α. As in [14], we can optimize the cost of this by checking many MACs in one go: we take a random linear combination of a and γ-values and check only the results of this. The full protocol is given in Figure 10; it is not intended to implement any functionality – it is just a procedure that can be called in both the offline and online phases. $\mathsf{MACCheck}$ has the following important properties.

Lemma 1. *The protocol* $\mathsf{MACCheck}$ *is* correct*, i.e. it accepts if all the values a_j and the corresponding MACs are correctly computed. Moreover, it is* sound*, i.e. it rejects except with probability $2/p$ in case at least one value or MAC is not correctly computed.*

The proof of Lemma 1 is given in Appendix D.3.

5.2 Offline Protocol

The offline phase itself runs two distinct sub-phases, each of which we now describe. To start with we assume a BGV key has been distributed according to the key generation procedure described earlier, as well as the shares of a secret MAC key and an encryption $\mathfrak{c}_\alpha$ of the MAC key as above. We assume that the output of the offline phase will be a total of at least n_I input tuples, n_m multiplication triples, n_s squaring tuples and n_b shared bits.

In the first sub-phase, which we call the tuple-production sub-phase, we over-produce the various multiplication and squaring tuples, plus the shared bits. These are then "sacrificed" in the tuple-checking phase so as to create at least n_m multiplication triples, n_s squaring tuples and n_b shared bits. In particular in the tuple-production phase we produce (at least) $2 \cdot n_m$ multiplication tuples, $2 \cdot n_s + n_b$ squaring tuples, and n_b shared bits. Tuple-production is performed by following the protocol in Figure 13 and Figure 14. The tuple production protocol can be run repeatedly, alongside the tuple-checking sub-phase and the online phase.

The second sub-phase of the offline phase is to check whether the resulting material from the prior phase has been produced correctly. This check is needed, because the distributed decryption procedure needed to produce the tuples and the MACs could allow the adversary to induce errors. We solve this problem via a sacrificing technique, as in [14], however, we also need to adapt it to the case of squaring tuples and bit-sharings. Moreover, this sacrificing is performed in the offline phase as opposed to the online phase (as in [14]); and the resulting partially opened values are checked in the offline phase (again as opposed to the online phase). This is made possible by our protocol MACCheck which allows to verify the MACs are correct without revealing the MAC key α. The tuple-checking protocol is presented in Figure 15.

We show that the resulting protocol Π_{PREP}, given in Figure 12, securely implements the functionality $\mathcal{F}_{\text{PREP}}$, which models the offline phase. The functionality $\mathcal{F}_{\text{PREP}}$ outputs some desired number of multiplication triples, squaring tuples and shared bits. In Appendix D we present a proof of the following theorem.

Theorem 4. *In the $(\mathcal{F}_{\text{SHE}}, \mathcal{F}_{\text{COMMIT}})$-hybrid model, the protocol Π_{PREP} implements $\mathcal{F}_{\text{PREP}}$ with computational security against any static adversary corrupting at most $n-1$ parties if p is exponential in the security parameter.*

The security flavour of Π_{PREP} follows the security of EncCommit, i.e. if one uses the covert (resp. active) version of EncCommit, one gets covert (resp. active) security for Π_{PREP}.

6 Online Phase

We design a protocol Π_{ONLINE} which performs the secure computation of the desired function, decomposed as a circuit over $\mathbb{F}_p$. Our online protocol makes use of the preprocessed data coming from $\mathcal{F}_{\text{PREP}}$ in order to input, add, multiply or square values. Our protocol is similar to the one described in [14]; however, it brings a series of improvements, in the sense that we could push the "sacrificing" to the preprocessing phase, we have specialised procedure for squaring etc, and we make use of a different MAC-checking method in the output phase. Our method for checking the MACs is simply the MACCheck protocol on all partially opened values; note that such a method has a lower soundness error than the method proposed in [14], since the linear combination of partially opened values is truly random in our case, while it has lower entropy in [14].

The following theorem, whose proof is given in Appendix E, shows that the protocol Π_{ONLINE}, given in Figure 20, securely implements the functionality $\mathcal{F}_{\text{ONLINE}}$, which models the online phase.

Theorem 5. *In the $\mathcal{F}_{\text{PREP}}$-hybrid model, the protocol Π_{ONLINE} implements $\mathcal{F}_{\text{ONLINE}}$ with computational security against any static adversary corrupting at most $n-1$ parties if p is exponential in the security parameter.*

The astute reader will be wondering where our shared bits produced in the offline phase are used. These will be used in "higher level" versions of the online phase (i.e. versions which do not just evaluate an arithmetic circuit) which implement the types of operations presented in [8, 9].

7 Experimental Results

7.1 KeyGen and Offline Protocols

To present performance numbers for our key generation and new variant of the offline phase for SPDZ we first need to define secure parameter sizes for the underlying BGV scheme (and in particular how it is used in our protocols). This is done in Appendix G, by utilizing the method of Appendices A.4, A.5 and B of [16], for various choices of n (the number of players) and p (the field size).

We then implemented the preceding protocols in C++ on top of the MPIR library for multi-precision arithmetic. Modular arithmetic was implemented with bespoke code using Montgomery arithmetic [20] and calls to the underlying `mpn_` functions in MPIR. The offline phase was implemented in a multi-threaded manner, with four cores producing initial multiplication triples, square pairs, shared bits and input preparation mask values. Then two cores performed the sacrificing for the multiplication triples, square pairs and shared bits.

In Table 1 we present execution times (in wall time measured in seconds) for key generation and for an offline phase which produces 100000 each of the multiplication tuples, square pairs, shared bits and 1000 input sharings. We also present the average time to produce a multiplication triple for an offline phase running on one core and producing 100000 multiplication triples only. The run-times are given for various values of n, p and c, and all timings were obtained on 2.80 GHz Intel Core i7 machines with 4 GB RAM, with machines running on a local network.

			Run Times		Time per
n	$p \approx$	c	KeyGen	Offline	Triple (sec)
2	2^{32}	5	2.4	156	0.00140
2	2^{32}	10	5.1	277	0.00256
2	2^{32}	20	10.4	512	0.00483
2	2^{64}	5	5.9	202	0.00194
2	2^{64}	10	12.5	377	0.00333
2	2^{64}	20	25.6	682	0.00634
2	2^{128}	5	16.2	307	0.00271
2	2^{128}	10	33.6	561	0.00489
2	2^{128}	20	74.5	1114	0.00937

			Run Times		Time per
n	$p \approx$	c	KeyGen	Offline	Triple(sec)
3	2^{32}	5	3.0	292	0.00204
3	2^{32}	10	6.4	413	0.00380
3	2^{32}	20	13.3	790	0.00731
3	2^{64}	5	7.7	292	0.00267
3	2^{64}	10	16.3	568	0.00497
3	2^{64}	20	33.7	1108	0.01004
3	2^{128}	5	21.0	462	0.00402
3	2^{128}	10	44.4	889	0.00759
3	2^{128}	20	99.4	2030	0.01487

Table 1. Execution Times For Key Gen and Offline Phase (Covert Security)

We compare the results to that obtained in [12], since no other protocol can provide malicious/covert security for $t < n$ corrupted parties. In the case of covert security the authors of [12] report figures of 0.002 seconds per (un-checked) 64-bit multiplication triple for both two and three players; however the probability of cheating being detected was lower bounded by $1/2$ for two players, and $1/4$ for three players; as opposed to our probabilities of $4/5$, $9/10$ and $19/20$. Since the triples in [12] were unchecked we need to scale their run-times by a factor of two; to obtain 0.004 seconds per multiplication triple. Thus for covert security we see that our protocol for checked tuples are superior both in terms error probabilities, for a comparable run-time.

When using our active security variant we aimed for a cheating probability of 2^{-40}; so as to be able to compare with prior run times obtained in [12], which used the method from [14]. Again we performed two experiments one where four cores produced 100000 multiplication triples, squaring pairs and shared bits, plus 1000 input sharings; and one experiment where one core produced just 100000 multiplication triples (so as to produce the average cost for a triple). The results are in Table 2.

By way of comparison for a prime of 64 bits the authors of [12] report on an implementation which takes 0.006 seconds to produce an (un-checked) multiplication triple for the case of two parties and equivalent active security; and 0.008 per second for the case of three parties and active security. As we produce checked triples, the cost per triple for the results in [12] need to be (at least) doubled; to produce a total of 0.012 and 0.016 seconds respectively.

Thus, in this test, our new active protocol has running time about twice that of the previous active protocol from [14] based on ZKPoKs. From the analysis of the protocols, we do expect that the new method will be faster, but only if we produce the output in large enough batches. Due to memory constraints we were so far unable to do this, but we can extrapolate from these results: In the test we generated 12 ciphertexts in one go, and if we were able to increase

	$n = 2$		$n = 3$	
$p \approx$	Offline	Time per Triple	Offline	Time per Triple
2^{32}	2366	0.01955	3668	0.02868
2^{64}	3751	0.02749	5495	0.04107
2^{128}	6302	0.04252	10063	0.06317

Table 2. Execution Times For Offline Phase (Active Security)

this by a factor of about 10, then we would get results better than those of [14, 12], all other things being equal. More information can be found in Appendix F.

7.2 Online

For the new online phase we have developed a purpose-built bytecode interpreter, which reads and executes pre-generated sequences of instructions in a multi-threaded manner. Our runtime supports parallelism on two different levels: independent rounds of communication can be merged together to reduce network overhead, and multiple threads can be executed at once to allow for optimal usage of modern multi-core processors.

Each bytecode instruction is either some local computation (e.g. addition of secret shared values) or an 'open' instruction, which initiates the protocol to reveal a shared value. The data from independent open instructions can be merged together to save on communication costs. Each player may run up to four different bytecode files in parallel in distinct threads, with each such thread having access to some shared memory resource. The advantage of this approach is that bytecode files can be pre-compiled and optimized, and then quickly loaded at runtime – the online phase runtime is itself oblivious to the nature of the programs being run.

In Table 3 we present timings (again in elapsed wall time for a player) for multiplying two secret shared values. Results are given for three different varieties of multiplication, reflecting the possibilities available: purely sequential multiplications; parallel multiplications with communication merged into one round (50 per round); and parallel multiplications running in 4 independent threads (50 per round, per thread). The experiments were carried out on the same machines as the offline phase, running over a local network with a ping of around 0.27ms. For comparison, the original implementation of the online phase in [14] gave an amortized time of 20000 multiplications per second over a 64-bit prime field, with three players.

		Multiplications/sec		
		Sequential	50 in Parallel	
n	$p \approx$	Single Thread	Single Thread	Four Threads
2	2^{32}	7500	134000	398000
2	2^{64}	7500	130000	395000
2	2^{128}	7500	120000	358000
3	2^{32}	4700	100000	292000
3	2^{64}	4700	98000	287000
3	2^{128}	4600	90000	260000

Table 3. Online Times

8 Acknowledgements

The first and fourth author acknowledge partial support from the Danish National Research Foundation and The National Science Foundation of China (under the grant 61061130540) for the Sino-Danish Center for the Theory of Interactive Computation, and from the CFEM research center (supported by the Danish Strategic Research Council). The second, third, fifth and sixth authors were supported by EPSRC via grant COED–EP/I03126X. The sixth author was also supported by the European Commission via an ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO,

the Defense Advanced Research Projects Agency and the Air Force Research Laboratory under agreement number FA8750-11-2-0079[3], and by a Royal Society Wolfson Merit Award.

References

1. M. Aliasgari, M. Blanton, Y. Zhang, and A. Steele. Secure computation on floating point numbers. In *Network and Distributed System Security Symposium – NDSS 2013*. Internet Society, 2013.
2. Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In *Theory of Cryptography – TCC 2007*, volume 4392 of *LNCS*, pages 137–156. Springer, 2007.
3. Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *J. Cryptology*, 23(2):281–343, 2010.
4. D. Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, volume 576 of *LNCS*, pages 420–432. Springer, 1991.
5. D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security – ESORICS 2008*, volume 5283 of *LNCS*, pages 192–206. Springer, 2008.
6. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *ITCS*, pages 309–325. ACM, 2012.
7. Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from Ring-LWE and security for key dependent messages. In *CRYPTO*, volume 6841 of *LNCS*, pages 505–524. Springer, 2011.
8. O. Catrina and A. Saxena. Secure computation with fixed-point numbers. In *Financial Cryptography – FC 2010*, volume 6052 of *LNCS*, pages 35–50. Springer, 2010.
9. 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 – TCC 2006*, volume 3876 of *LNCS*, pages 285–304. Springer, 2006.
10. I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous multiparty computation: Theory and implementation. In *Public Key Cryptography – PKC 2009*, volume 5443 of *LNCS*, pages 160–179. Springer, 2009.
11. I. Damgård and M. Keller. Secure multiparty AES. In *Financial Cryptography*, volume 6052 of *LNCS*, pages 367–374. Springer, 2010.
12. I. Damgård, M. Keller, E. Larraia, C. Miles, and N. P. Smart. Implementing aes via an actively/covertly secure dishonest-majority mpc protocol. In *SCN*, volume 7485 of *LNCS*, pages 241–263. Springer, 2012.
13. I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure MPC for dishonest majority – or: Breaking the SPDZ limits, 2012.
14. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, 2012.
15. C. Gentry, S. Halevi, and N. P. Smart. Fully homomorphic encryption with polylog overhead. In *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 465–482. Springer, 2012.
16. C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *LNCS*, pages 850–867. Springer, 2012.
17. B. Kreuter, A. Shelat, and C.-H. Shen. Towards billion-gate secure computation with malicious adversaries. In *USENIX Security Symposium – 2012*, pages 285–300, 2012.
18. Y. Lindell, B. Pinkas, and N. P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In *Security and Cryptography for Networks – SCN 2008*, volume 5229 of *LNCS*, pages 2–20. Springer, 2008.
19. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - Secure two-party computation system. In *USENIX Security Symposium – 2004*, pages 287–302, 2004.
20. P. L. Montgomery. Modular multiplication without trial division. *Math. Comp.*, 44:519–521, 1985.
21. J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, 2012.
22. J. B. Nielsen and C. Orlandi. LEGO for two-party secure computation. In *TCC*, volume 5444 of *LNCS*, pages 368–386. Springer, 2009.
23. C. Peikert, V. Vaikuntanathan, and B. Waters. A framework for efficient and composable oblivious transfer. *Advances in Cryptology–CRYPTO 2008*, pages 554–571, 2008.

[3] The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, AFRL, or the U.S. Government.

24. B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, 2009.
25. SIMAP Project. SIMAP: Secure information management and processing. `http://alexandra.dk/uk/Projects/Pages/SIMAP.aspx`.

A Commitments in the Random Oracle Model

A.1 Protocol and Functionality

The Protocol Π_{COMMIT}

Commit:

1. In order to commit to v, P_i sets $o \leftarrow v||r$, where r is chosen uniformly in a determined domain, and queries the Random Oracle $\mathcal{H}$ to get $c \leftarrow \mathcal{H}(o)$.
2. P_i then broadcasts (c, i, τ_v), where τ_v represents a handle for the commitment.

Open:

1. In order to open a commitment (c, i, τ_v), where $c = \mathcal{H}(v||r)$, player P_i broadcasts $(o = v||r, i, \tau_v)$.
2. All players call $\mathcal{H}$ on o and check whether $\mathcal{H}(o) = c$. Players accept if and only if this check passes.

Fig. 1. The Protocol for Commitments.

The Ideal Functionality $\mathcal{F}_{\text{COMMIT}}$

Commit: On input $(\mathsf{Commit}, v, i, \tau_v)$ by P_i or the adversary on his behalf (if P_i is corrupt), where v is either in a specific domain or $\perp$, it stores (v, i, τ_v) on a list and outputs (i, τ_v) to all players and adversary.

Open: On input $(\mathsf{Open}, i, \tau_v)$ by P_i or the adversary on his behalf (if P_i is corrupt), the ideal functionality outputs (v, i, τ_v) to all players and adversary. If $(\mathsf{NoOpen}, i, \tau_v)$ is given by the adversary, and P_i is corrupt, the functionality outputs $(\perp, i, \tau_v)$ to all players.

Fig. 2. The Ideal Functionality for Commitments

A.2 UC Security

Lemma 2. *In the random oracle model, the protocol Π_{COMMIT} implements $\mathcal{F}_{\text{COMMIT}}$ with computational security against any static, active adversary corrupting at most $n - 1$ parties.*

Proof. We here sketch a simulator such that the environment cannot distinguish if it is playing with the real protocol or the functionality composed with the simulator. Note that the simulator replies to queries to the random oracle $\mathcal{H}$ made by the adversary.

To simulate a Commit call, if the committer P_i is honest, the simulator selects a random value c and gives (c, i, τ_v) to the adversary. Note that (i, τ_v) is given to the simulator by $\mathcal{F}_{\text{COMMIT}}$ hereafter receiving $(\mathsf{Commit}, v, i, \tau_v)$ from P_i. Whereas if the committer is corrupt, then it either queries $\mathcal{H}$ to get c, or it does not query it. Therefore, on receiving (c^*, i, τ_v) from the adversary, the simulator has v (if $\mathcal{H}$ was queried) so it sets $v^* \leftarrow v$. If $\mathcal{H}$ was not queried, the simulator sets dummy input v^* and the internal flag $\mathsf{Abort}_{i,\tau_v}$ to true. It then sends $(\mathsf{Commit}, v^*, i, \tau_v)$ to $\mathcal{F}_{\text{COMMIT}}$.

An Open call is simulated as follows. If the committer is honest, the simulator gets (v, i, τ_v) when P_i inputs $(\mathsf{Open}, i, \tau_v)$ to $\mathcal{F}_{\text{COMMIT}}$. The simulator selects random r and sets $o \leftarrow v||r$. It can now hand (o, i, τ_v) to the adversary. If the random oracle is queried on o, the simulator sends c as response. If the committer is corrupt, the simulator gets (i, τ_v) from the adversary, it checks whether $\mathsf{Abort}_{i,\tau_v}$ is true, if so it sends $(\mathsf{NoOpen}, i, \tau_v)$ to $\mathcal{F}_{\text{COMMIT}}$. Otherwise, the simulator sends $(\mathsf{Open}, i, \tau_i)$ to $\mathcal{F}_{\text{COMMIT}}$.

The adversary will notice that queries to $\mathcal{H}$ are simulated only if o has been queried before resulting in different c, but as r is random this happens only with negligible probability (assuming that the size of the output domain of $\mathcal{H}$ is large enough). Also, in a simulated run, if the adversary does not query $\mathcal{H}$ when committing will result in abort. The

probability that in a real run players do not abort, is equivalent to the the probability that adversary correctly guesses the output of $\mathcal{H}$, which happens with negligible probability.

□

B Key Generation : Protocol, Functionalities and Security Proof

B.1 Protocol

The protocol Π_{KEYGEN}

Initialize:

1. Every player P_i samples a uniform $e_i \leftarrow \{1, \ldots, c\}$ and asks $\mathcal{F}_{\text{COMMIT}}$ to broadcast the handle $\tau_i^e \leftarrow \mathsf{Commit}(e_i)$ for a commitment to e_i.
2. Every player P_i samples a seed $s_{i,j}$ and asks $\mathcal{F}_{\text{COMMIT}}$ to broadcast $\tau_{i,j}^s \leftarrow \mathsf{Commit}(s_{i,j})$.
3. Every player P_i computes and broadcasts $a_{i,j} \leftarrow \mathcal{U}_{s_{i,j}}(q_1, \phi(m))$.

Stage 1:

4. All the players compute $a_j \leftarrow a_{1,j} + \cdots + a_{n,j}$.
5. Every player P_i computes $\mathfrak{s}_{i,j} \leftarrow \mathcal{HWT}_{s_{i,j}}(h, \phi(m))$ and $\epsilon_{i,j} \leftarrow \mathcal{DG}_{s_{i,j}}(\sigma^2, \phi(m))$, and broadcasts $b_{i,j} \leftarrow [a_j \cdot \mathfrak{s}_{i,j} + p \cdot \epsilon_{i,j}]_{q_1}$.

Stage 2:

6. All the players compute $b_j \leftarrow b_{1,j} + \cdots + b_{n,j}$ and set $\mathfrak{pk}_j \leftarrow (a_j, b_j)$..
7. Every player P_i computes and broadcasts $\mathfrak{enc}'_{i,j} \leftarrow \mathsf{Enc}_{\mathfrak{pk}_j}(-p_1 \cdot \mathfrak{s}_{i,j}, \mathcal{RC}_{s_{i,j}}(0.5, \sigma^2, \phi(m)))$.

Stage 3:

8. All the players compute $\mathfrak{enc}'_j \leftarrow \mathfrak{enc}'_{1,j} + \cdots + \mathfrak{enc}'_{n,j}$.
9. Every player P_i computes $\mathfrak{zero}_{i,j} \leftarrow \mathsf{Enc}_{\mathfrak{pk}_j}(0, \mathcal{RC}_{s_{i,j}}(0.5, \sigma^2, \phi(m)))$.
10. Every player P_i computes and broadcasts $\mathfrak{enc}_{i,j} \leftarrow (\mathfrak{s}_{i,j} \cdot \mathfrak{enc}'_j) + \mathfrak{zero}_{i,j}$.

Output:

11. All the players compute $\mathfrak{enc}_j \leftarrow \mathfrak{enc}_{1,j} + \cdots + \mathfrak{enc}_{n,j}$ and set $\mathfrak{epk}_j \leftarrow (\mathfrak{pk}_j, \mathfrak{enc}_j)$.
12. Every player P_i calls $\mathcal{F}_{\text{COMMIT}}$ with $\mathsf{Open}(\tau_i^e)$. If any opening failed, the players output the numbers of the respective players, and the protocol aborts.
13. All players compute the challenge $\mathsf{chall} \leftarrow 1 + \left(\left(\sum_{i=1}^n e_i\right) \bmod c\right)$.
14. Every player P_i calls $\mathcal{F}_{\text{COMMIT}}$ with $\mathsf{Open}(\tau_{i,j}^s)$ for $j \neq \mathsf{chall}$. If any opening failed, the players output the numbers of the respective players, and the protocol aborts.
15. All players obtain the values committed, compute all the derived values and check that they are correct.
16. If any of the checks fail, the players output the numbers of the respective players, and the protocol aborts. Otherwise, every player P_i sets
    - $\mathfrak{s}_i \leftarrow \mathfrak{s}_{i,\mathsf{chall}}$,
    - $\mathfrak{pk} \leftarrow (a_{\mathsf{chall}}, b_{\mathsf{chall}})$, $\mathfrak{epk} \leftarrow (\mathfrak{pk}, \mathfrak{enc}_{\mathsf{chall}})$.

Fig. 3. The protocol for key generation.

B.2 Functionality

The ideal functionality $\mathcal{F}_{\text{KEYGEN}}$

Generation:

1. The functionality waits for seeds $\{s_i\}_{i\in A}$ from the adversary, and samples a seed s_i for every honest player P_i.
2. It computes $a_i \leftarrow \mathcal{U}_{s_i}(q_1, \phi(m))$ for all i.
3. It computes $\mathfrak{s}_i \leftarrow \mathcal{HWT}_{s_i}(h, \phi(m))$, $\epsilon_i \leftarrow \mathcal{DG}_{s_i}(\sigma^2, \phi(m))$ and $b_i \leftarrow [a \cdot \mathfrak{s}_i + p \cdot \epsilon_i]_{q_1}$ for all i.
4. It computes $b \leftarrow b_1 + \cdots + b_n$.
5. It computes $\mathfrak{enc}'_i \leftarrow \mathsf{Enc}_{\mathfrak{pk}}(-p_1 \cdot \mathfrak{s}_i, \mathcal{RC}_{s_i}(0.5, \sigma^2, \phi(m)))$ for every P_i.
6. It computes $\mathfrak{enc}' \leftarrow \mathfrak{enc}'_1 + \cdots + \mathfrak{enc}'_n$.
7. It computes $\mathfrak{zero}_i \leftarrow \mathsf{Enc}_{\mathfrak{pk}}(0, \mathcal{RC}_{s_i}(0.5, \sigma^2, \phi(m)))$ for every P_i, and $\mathfrak{enc}_i \leftarrow (\mathfrak{s}_i \cdot \mathfrak{enc}') + \mathfrak{zero}_i$ for all players P_i.
8. It leaks $a_i, b_i, \mathfrak{enc}'_i, \mathfrak{enc}_i$ for all i to the adversary and waits for either Proceed, Cheat, or Abort.

Proceed: The functionality sends $\mathfrak{s}_i$, $\mathfrak{pk} = (a, b)$, $\mathfrak{epk} = (\mathfrak{pk}, \mathfrak{enc})$ to P_i, for all i.

Cheat: On input Cheat, with probability $1 - 1/c$ the functionality leaks NoSuccess and goes to "Abort"; otherwise:

1. It leaks the seeds of the honest parties and sends Success to the adversary.
2. It repeats the following loop:
   - It waits for the adversary to input values a_i^*, (resp. $(\mathfrak{s}_i^*, b_i^*)$, $\mathfrak{enc}_i'^*$, $\mathfrak{enc}_i^*$) for $i \in A$.
   - It overwrites a_i, (resp. $(\mathfrak{s}_i, b_i)$, $\mathfrak{enc}'_i$, $\mathfrak{enc}_i$) for $i \in A$.
   - It recomputes a (resp. b, $\mathfrak{enc}'$, $\mathfrak{enc}$) accordingly.
3. It waits for Proceed or Abort.

Abort:

1. The functionality leaks the seeds of the honest parties if it never did so.
2. It then waits for a set $S \subseteq A$, sends it to the honest players, and aborts.

Fig. 4. The ideal functionality for key generation.

B.3 Proof of Theorem 1

Proof. We build a simulator $\mathcal{S}_{\text{KEYGEN}}$ to work on top of the ideal functionality $\mathcal{F}_{\text{KEYGEN}}$, such that the environment cannot distinguish whether it is playing with the protocol Π_{KEYGEN} and $\mathcal{F}_{\text{COMMIT}}$, or the simulator and $\mathcal{F}_{\text{KEYGEN}}$. The simulator is given in Figure 6.

We now proceed with the analysis of the simulation. Let A denote the set of players controlled by the adversary. In steps 1 and 2 the simulator sends random handles to the adversary, as it would happen in the real protocol. In steps 3–11 the simulation is perfect for all the threads where the simulator knows the seeds of the honest players, since those are generated as in the protocol. In case of no cheat nor abort the simulation is also perfect for the thread where the simulator does not know the seeds of the honest players, since the simulator forwards honest values provided by the functionality. In case of cheating at the thread pointed by chall, the simulator gets the seeds also for the remaining thread and will replace the honestly precomputed intermediate values $a_{i,\mathsf{chall}}$, $\mathfrak{s}_{i,\mathsf{chall}}$, $b_{i,\mathsf{chall}}$, $\mathfrak{enc}'_{i,\mathsf{chall}}$, $\mathfrak{enc}_{i,\mathsf{chall}}$ with the ones compatible with the deviation of the adversary – the honest values computed after a cheat reflect the adversarial behaviour of the real protocol, so a simulated run is again indistinguishable from a real run of the protocol.

Steps 12, 14 are statistically indistinguishable from a protocol run, since the simulator plays also the role of $\mathcal{F}_{\text{COMMIT}}$.

Step 15 needs more work: we need to ensure that the success probability in a simulated run is the same as the one in a real run of the protocol. If the adversary does not deviate, the protocol succeeds. The same applies for a simulated run, since the simulator goes through "Pass" at every stage. More in detail, the simulator sampled and computed all the values at the threads not pointed by the challenge as in a honest run of the protocol, while values at the thread pointed by the challenge are correctly evaluated and sent to the honest players by $\mathcal{F}_{\text{KEYGEN}}$. In case the adversary cheats only on one thread, in a real execution of the protocol the adversary succeeds in the protocol with probability $1/c$; the same holds in a simulated run, since the simulator goes through Cheat in CheatSwitch once and the functionality leaks

Success, which happens with the same probability, and later the simulator will not abort. If the adversary deviates on more than one branch (considering all stages), both the real protocol and the simulation will abort at step 15.

Finally, if the protocol aborts due to failure at opening commitments, both the functionality and the players output the numbers of corrupted players who failed to open their commitments. If the protocol aborts at step 15, the output is the numbers of players who deviated in threads other than chall in both the functionality and the protocol. □

The procedure CheatSwitch

Pass: Checks passed for all threads or there was a successful cheat in earlier stages and checks now passed for at least all threads except for the one pointed by chall.
- The simulator continues.

Cheat: There was no cheat in previous stages and all checks now passed except for the ones pointed by a unique thread j.
- The simulator adds (j, P_i) to a list L, for all players P_i making the check not pass.
- The simulator sends Cheat to $\mathcal{F}_{\text{KEYGEN}}$, and gets and stores every honest seeds $s_{i,\mathsf{chall}}$.
- If the functionality sends NoSuccess:
  - If $j = \mathsf{chall}$, it resamples a different $\mathsf{chall} \leftarrow \{1, \ldots, c\} \setminus \{\mathsf{chall}\}$.

  It continues the simulation according to the protocol.
- If the functionality sends Success: the simulator sets $\mathsf{chall} = j$ and continues according to the protocol.

Abort: In more than one thread checks did not pass (counting also checks in previous stages).
- The simulator adds (j, P_i) to a list L, for all branches j and players P_i making the check not pass.
- The simulator sends Abort to $\mathcal{F}_{\text{KEYGEN}}$ and it continues the rest of the simulation according to the protocol.

Fig. 5. The cheat switch.

The simulator $\mathcal{S}_{\text{KEYGEN}}$

Initialize:

- In Step 1 the simulator obtains e_i by every corrupt P_i, and broadcasts τ_i^e as $\mathcal{F}_{\text{COMMIT}}$ would do. It samples chall uniformly in $\{1,\dots,c\}$, and it broadcasts a handle τ_i^e for every honest P_i.
- In Step 2 the simulator sees the random values $s_{i,j}$ for $i \in A$.
  It inputs $\{s_{i,\mathsf{chall}}\}_{i\in A}$ to $\mathcal{F}_{\text{KEYGEN}}$, therefore obtaining a full transcript of the thread corresponding to chall.
  For the threads $j \neq \mathsf{chall}$, for honest P_i, the simulator samples $s_{i,j}$ honestly and broadcasts a handle $\tau_{i,j}^s$ for every honest P_i for every thread.
- In Step 3, the simulator computes $a_{i,j}$ honestly for $i \notin A$ and $j \neq \mathsf{chall}$, while it defines $a_{i,\mathsf{chall}}$ for $i \notin A$ as the values a_i obtained from the transcript given by $\mathcal{F}_{\text{KEYGEN}}$.
  It then broadcasts $a_{i,j}$ for honest P_i and waits for broadcasts $a_{i,j}$ by the corrupt players, and it checks $a_{i,j} = \mathcal{U}_{s_{i,j}}(q, \phi(m))$ for all dishonest P_i. For this check the simulator enters $\mathsf{CheatSwitch}$. If there was a successful cheat on the thread pointed by chall, the simulator inputs $a_{i,\mathsf{chall}}$ to $\mathcal{F}_{\text{KEYGEN}}$ for $i \in A$.

Stage 1:

- In Step 4 the simulator acts as in the protocol.
- In Step 5 for all the honest seeds that are known by the simulator, the simulator computes $b_{i,j}$ honestly for $i \notin A$.
  If the simulator does not know the seeds $s_{i,\mathsf{chall}}$ for honest P_i, it defines $b_{i,\mathsf{chall}}$ for $i \notin A$ as the values b_i obtained from the transcript given by $\mathcal{F}_{\text{KEYGEN}}$.
  It then broadcasts $b_{i,j}$ for honest P_i and waits for broadcasts $b_{i,j}$ by the corrupt players. It then checks $b_{i,j} = [a_{\mathsf{chall}} \cdot \mathcal{HWT}_{s_{i,\mathsf{chall}}}(h, \phi(m)) + p \cdot \mathcal{RC}_{s_{i,\mathsf{chall}}}(\sigma^2, \phi(m))]_{q_1}$ for all dishonest P_i. For this check the simulator enters $\mathsf{CheatSwitch}$. If there was a successful cheat on the thread pointed by chall, the simulator inputs $(\mathfrak{s}_{i,\mathsf{chall}}, b_{i,\mathsf{chall}}$ to $\mathcal{F}_{\text{KEYGEN}}$ for $i \in A$.

Stage 2:

- In Step 6 the simulator acts as in the protocol.
- In Step 7 for all the honest seeds that are known by the simulator, the simulator computes $\mathfrak{enc}'_{i,j}$ honestly for $i \notin A$.
  If the simulator does not know the seeds $s_{i,\mathsf{chall}}$ for honest P_i, it defines $\mathfrak{enc}'_{i,\mathsf{chall}}$ for $i \notin A$ as the values $\mathfrak{enc}'_i$ obtained from the transcript given by $\mathcal{F}_{\text{KEYGEN}}$.
  It then broadcasts $\mathfrak{enc}'_{i,j}$ for honest P_i and waits for broadcasts $\mathfrak{enc}'_{i,j}$ by the corrupt players. It then checks $\mathfrak{enc}'_{i,j} = \mathsf{Enc}_{\mathfrak{pk}}(-p_1 \cdot \mathfrak{s}_{i,j}, \mathcal{RC}_{s_{i,j}}(0.5, \sigma^2, \phi(m)))$ for all dishonest P_i. For this check the simulator enters $\mathsf{CheatSwitch}$. If there was a successful cheat on the thread pointed by chall, the simulator inputs $\mathfrak{enc}'_{i,\mathsf{chall}}$ to $\mathcal{F}_{\text{KEYGEN}}$ for $i \in A$.

Stage 3:

- In Step 8, 9 the simulator acts as in the protocol.
- In Step 10 for all the honest seeds that are known by the simulator, the simulator computes $\mathfrak{enc}_{i,j}$ honestly for $i \notin A$.
  If the simulator does not know the seeds $s_{i,\mathsf{chall}}$ for honest P_i, it defines $\mathfrak{enc}_{i,\mathsf{chall}}$ for $i \notin A$ as the values $\mathfrak{enc}_i$ obtained from the transcript given by $\mathcal{F}_{\text{KEYGEN}}$.
  It then broadcasts $\mathfrak{enc}_{i,j}$ for honest P_i and waits for broadcasts $\mathfrak{enc}_{i,j}$ by the corrupt players. It then checks $\mathfrak{enc}_{i,j} = (\mathfrak{s}_{i,j} \cdot \mathfrak{enc}'_{i,j}) + \mathfrak{zero}_{i,j}$ for all dishonest P_i. For this check the simulator enters $\mathsf{CheatSwitch}$. If there was a successful cheat on the thread pointed by chall, the simulator inputs $\mathfrak{enc}_{i,\mathsf{chall}}$ to $\mathcal{F}_{\text{KEYGEN}}$ for $i \in A$.

Output:

- Step 11 is performed according to the protocol.
- The simulator samples e_i for $i \notin A$ uniformly such that $1 + \left(\left(\sum_{i=1}^n e_i\right) \bmod c\right) = \mathsf{chall}$.
- Step 12 is performed according to the protocol, but the simulator opens τ_i^e revealing the values e_i for all honest P_i, and if the check fails the simulator sends Abort to $\mathcal{F}_{\text{KEYGEN}}$ and inputs the set of all players failing in opening.
- Step 13 is performed according to the protocol.
- Step 14 is performed according to the protocol, and if the check fails the simulator sends Abort to $\mathcal{F}_{\text{KEYGEN}}$ and inputs the set of all players failing in opening.
- Step 15 is performed according to the protocol, and the simulator defines

$$S = \{i \in A \mid (j, P_i) \in L;\ j \in \{1,\dots,c\};\ j \neq \mathsf{chall}\},$$

  i.e. the set of corrupt players who cheated at any thread different from chall.
  - If $S \neq \varnothing$ (i.e. cheats at a thread which is going to be opened), the simulator sends Abort to $\mathcal{F}_{\text{KEYGEN}}$ and inputs S.
  - If $S = \varnothing$ (i.e. successful or no cheats), the simulator sends $\mathsf{Proceed}$ to $\mathcal{F}_{\text{KEYGEN}}$.
- Step 16 is performed according to the protocol.

Fig. 6. The simulator for the key generation functionality.

B.4 Semantic security of $\mathcal{F}_{\text{KEYGEN}}$

Here we prove the semantic security of the cryptosystem resulting from an execution of $\mathcal{F}_{\text{KEYGEN}}$, based on the ring-LWE problem and a form of KDM security for quadratic functions. The ring-LWE assumption we use takes an extra parameter h, as our scheme chooses binary, low hamming weight, secret keys for better efficiency and parameter sizes, but note that the results here also apply to secrets drawn from other distributions.

Definition 1 (Decisional Ring Learning With Errors assumption). *The single sample decisional ring-LWE assumption* $\mathsf{RLWE}_{q,\sigma^2,h}$ *states that*

$$(a, a \cdot s + e) \overset{c}{\approx} (a, u)$$

where $s \leftarrow \mathcal{HWT}(h, \phi(m))$*,* $e \leftarrow \mathcal{DG}(\sigma^2, \phi(m))$ *and* a, u *are uniform over* R_q*.*

The KDM security assumption, below, can be viewed as a distributed extension to the usual key switching assumption for FHE schemes. In this case we need 'encryptions' of quadratic functions of *additive shares* of the secret key to remain secure. Note that whilst it is easy to show KDM security for *linear* functions of the secret [7], it is not known how to extend this to the functions required here without increasing the length of ciphertexts.

Definition 2 (KDM security assumption). *If* $\mathfrak{s}_i \leftarrow \mathcal{HWT}(h)$*,* $\mathfrak{s} = \sum_{i=0}^{n-1} \mathfrak{s}_i$ *and* f *is any degree 2 polynomial then*

$$(a, a \cdot \mathfrak{s} + p \cdot e + f(\mathfrak{s}_0, \ldots, \mathfrak{s}_{n-1})) \overset{c}{\approx} (a, a \cdot \mathfrak{s} + p \cdot e)$$

where $a, u \leftarrow \mathcal{U}(q, \phi(m))$*,* $e \leftarrow \mathcal{DG}(\sigma^2, \phi(m))$*.*

The following lemma states that distinguishing any number of 'amortized' ring-LWE samples with different, independent, secret keys but common first component a, from uniform is as hard as distinguishing just one ring-LWE sample from uniform. It was proven for the (standard) LWE setting with $n = 3$ in [23]; here we need a version with ring-LWE for any n.

Lemma 3 (Adapted from [23, Lemma 7.6]). *Suppose* $a, u_i \leftarrow \mathcal{U}(q, \phi(m))$*,* $\mathfrak{s}_i \leftarrow \mathcal{HWT}(h, \phi(m))$ *and* $e_i \leftarrow \mathcal{DG}(\sigma^2, \phi(m))$ *for* $i = 0, \ldots, n-1$*,* $n \in \mathbb{N}$*. Then*

$$\{(a, a \cdot \mathfrak{s}_i + e_i)\}_i \overset{c}{\approx} \{(a, u_i)\}_i$$

under the single sample ring-LWE assumption $\mathsf{RLWE}_{q,\sigma^2,h}$*.*

Proof. Suppose an adversary $\mathcal{A}$ can distinguish between the above distributions with non-negligible probability. We construct an adversary $\mathcal{B}$ that solves the RLWE problem. Given a challenge (a, b) from the RLWE oracle, $\mathcal{B}$ sets $b_0 = b$ and $b_i = a \cdot \mathfrak{s}_i + e_i$ for $i = 1, \ldots, n-1$, where $e_i \leftarrow \mathcal{DG}(\sigma^2, \phi(m))$, $\mathfrak{s}_i \leftarrow \mathcal{HWT}(h, \phi(m))$. $\mathcal{B}$ sends all pairs (a, b_i) to $\mathcal{A}$ and returns the output of $\mathcal{A}$ in response to the challenge.

Since the values (a, b_i) for $i = 1, \ldots, n-1$ are all valid amortized ring-LWE samples, the only difference between the view of $\mathcal{A}$ and that of a real set of inputs is b_0, and so the advantage of $\mathcal{B}$ in solving $\mathsf{RLWE}_{q,\sigma^2,h}$ is exactly that of $\mathcal{A}$ in solving the amortized ring-LWE problem with n samples. □

Theorem 6 (restatement of Theorem 2). *If the functionality* $\mathcal{F}_{\text{KEYGEN}}$ *is used to produce a public key* $\mathfrak{epk}$ *and secret keys* $\mathfrak{s}_i$ *for* $i = 0, \ldots, n-1$ *then the resulting cryptosystem is semantically secure based on the hardness of* $\mathsf{RLWE}_{q_1,\sigma^2,h}$ *and the KDM security assumption.*

Proof. Suppose there is an adversary $\mathcal{A}$ that can interact with $\mathcal{F}_{\text{KEYGEN}}$ and distinguish the public key $(\mathfrak{pk}, \mathfrak{epk})$ from uniform. We construct an algorithm $\mathcal{B}$ that distinguishes amortized ring-LWE samples from uniform. By Lemma 3 this is at least as hard as breaking single sample ring-LWE. If the public key is pseudorandom then semantic security of encryption easily follows, as ciphertexts are just ring-LWE samples. Note that we only consider a non-cheating adversary – if $\mathcal{A}$ cheats then it can trivially break the scheme with non-negligible probability $1/c$.

The challenger gives $\mathcal{B}$ the values $a_c, b_{c,0}, \ldots, b_{c,n-1}$. $\mathcal{B}$ must now simulate an execution of $\mathcal{F}_{\text{KEYGEN}}$ with $\mathcal{A}$ to determine whether the challenge is uniform or of the form $(a_c, a_c \cdot \mathfrak{s}_i + e_i)$ for $\mathfrak{s}_i \leftarrow \mathcal{HWT}(h, \phi(m))$ and $e_i \leftarrow \mathcal{DG}(\sigma^2, \phi(m))$.

To start with we receive the adversary's seeds s_i for every corrupt player $i \in A$. We must then simulate the values $a_i, b_i, \mathfrak{enc}'_i, \mathfrak{enc}_i$ (for all i) that are leaked to the adversary in $\mathcal{F}_{\text{KEYGEN}}$. For corrupt players we simply compute these values according to $\mathcal{F}_{\text{KEYGEN}}$ using the adversary's seeds. Next we have to simulate the honest players' values, which we do using the challenge $a_c, b_{c,0}, \ldots, b_{c,n-1}$. First we scale the challenge by p, so that it takes the form $(a_c, a_c \cdot \mathfrak{s}_i + p \cdot e_i)$ if they are genuine RLWE samples. Since p is coprime to q this still has the same distribution as the original challenge.

Now $\mathcal{B}$ calculates uniform consistent shares $a_{c,i}$, for every honest player P_i, of a_c, and sends $\mathcal{A}$ the pairs $a_{c,i}, b_{c,i}$. If the challenge values are amortized ring-LWE samples, then these are consistent with the pairs (a_i, b_i) computed by $\mathcal{F}_{\text{KEYGEN}}$, since a_i is uniform and $b_i = a_i \cdot \mathfrak{s}_i + e_i$.

Next, $\mathcal{B}$ must provide $\mathcal{A}$ with simulations of players' contributions to the key-switching data $\mathfrak{enc}'_i, \mathfrak{enc}_i$ for all honest players P_i. For both of these sets of values, $\mathcal{B}$ simply re-randomizes the pair (a_c, b_c) and sends this to $\mathcal{A}$. This can be done by, for example, computing an encryption of zero under the public key (a_c, b_c) (where $b_c = \sum_i b_{c,i}$). Notice that $\mathfrak{enc}'_i$ is just an encryption of $-p_1 \cdot \mathfrak{s}_i$ under the public key (a, b), and so by the KDM security assumption is (perfectly) indistinguishable from a re-randomized version of (a, b). For $\mathfrak{enc}_i$, recall that $\mathcal{F}_{\text{KEYGEN}}$ computes $\mathfrak{enc}_i = \mathfrak{s}_i \cdot \mathfrak{enc}' + \mathfrak{zero}_i$. Now writing $\mathfrak{zero}_i = (a \cdot v_i + p \cdot e_{0,i}, b \cdot v_i + p \cdot e_{1,i})$ and $\mathfrak{enc}' = (a \cdot v + p \cdot e_0, b \cdot v + p \cdot e_1 - p_1 \cdot \mathfrak{s})$, we see that

$$
\begin{aligned}
\mathfrak{enc}_i &= (a \cdot v \cdot \mathfrak{s}_i + a \cdot v_i + p \cdot (e_0 \cdot \mathfrak{s}_i + e_{0,i}), b \cdot v \cdot \mathfrak{s}_i + b \cdot v_i + p \cdot (e_1 \cdot \mathfrak{s}_i + e_{1,i}) - p_1 \cdot \mathfrak{s} \cdot \mathfrak{s}_i) \\
&= \left(\underbrace{a \cdot (v \cdot \mathfrak{s}_i + v_i)}_{a'_i} + p \cdot \underbrace{(e_0 \cdot \mathfrak{s}_i + e_{0,i})}_{e'_{0,i}}, \underbrace{a \cdot (v \cdot \mathfrak{s}_i + v_i)}_{a'_i} \cdot \mathfrak{s} + p \cdot \underbrace{(e_1 \cdot \mathfrak{s}_i + e_{1,i} + e)}_{e'_{1,i}} - p_1 \cdot \mathfrak{s} \cdot \mathfrak{s}_i \right) \\
&= \left(a'_i + p \cdot e'_{0,i}, a'_i \cdot \mathfrak{s} + p \cdot e'_{1,i} - p_1 \cdot \mathfrak{s} \cdot \mathfrak{s}_i \right).
\end{aligned}
$$

Notice that the first component of $\mathfrak{enc}_i$ corresponds to the second half of a ring-LWE sample $(a, a \cdot (v \cdot \mathfrak{s}_i + v_i) + p \cdot e_{0,i})$ with secret $v \cdot \mathfrak{s}_i + v_i$. The second component of $\mathfrak{enc}_i$ corresponds to a ring-LWE sample with secret $\mathfrak{s}$ and first half a'_i, with an added quadratic function of the key $-p_1 \cdot \mathfrak{s} \cdot \mathfrak{s}_i$. By the KDM security assumption, this is indistinguishable from a genuine ring-LWE sample, so $\mathfrak{enc}_i$ can also be perfectly simulated by re-randomizing (a_c, b_c).

To finish the simulated execution of $\mathcal{F}_{\text{KEYGEN}}$, $\mathcal{B}$ sends $\mathcal{A}$ shares of the secret key for all P_i where $i \in A$ (i.e. all dishonest players), by sampling randomness using the seeds that were provided to $\mathcal{B}$ at the beginning. $\mathcal{B}$ then waits for $\mathcal{A}$ to give an answer and returns this in response to the challenger. Notice that throughout the simulation, all values passed to $\mathcal{A}$ were ring-LWE samples derived from the challenge $(a_c, b_{0,c}, \ldots, b_{n-1,c})$. We showed that if the challenge is an amortized ring-LWE sample then $\mathcal{A}$'s input is indistinguishable from the output of $\mathcal{F}_{\text{KEYGEN}}$, whereas if the challenge is uniform then so is $\mathcal{A}$'s input. Therefore if $\mathcal{A}$ is successful in distinguishing the resulting public key from uniform then $\mathcal{A}$ must have solved the ring-LWE challenge.

□

C EncCommit: Protocol, Functionalities and Security Proofs

C.1 Protocol

Protocol $\Pi_{\text{EncCommit}}$

Usage: The specific distribution of the message is defined by the input parameter cond. The output is a single message $\mathbf{m}_i$ private to each player, and a public ciphertext $\mathfrak{c}_i$ from player i. The protocol runs in two phases; a commitment phase and an opening phase.

KeyGen: The players execute Π_{KeyGen} to obtain $\mathfrak{s}_i$, $\mathfrak{pk}$, and $\mathfrak{epk}$.

Commitment Phase:

1. Every player P_i samples a uniform $e_i \leftarrow \{1, \ldots, c\}$, and queries $\mathsf{Commit}(e_i)$ to $\mathcal{F}_{\text{Commit}}$, which broadcasts a handle τ_i^e.
2. For $j = 1, \ldots, c$
   (a) Every player P_i samples a seed $s_{i,j}$ and queries $\mathsf{Commit}(s_{i,j})$ to $\mathcal{F}_{\text{Commit}}$, which broadcasts a handle $\tau_{i,j}^s$.
   (b) Every player P_i generates $\mathbf{m}_{i,j}$ according to cond using $\mathsf{PRF}_{s_{i,j}}$.
   (c) Every player P_i computes and broadcasts $\mathfrak{c}_{i,j} \leftarrow \mathsf{Enc}_{\mathfrak{pk}}(\mathbf{m}_{i,j})$ using $\mathsf{PRF}_{s_{i,j}}$ to generate the randomness.
3. Every player P_i calls $\mathcal{F}_{\text{Commit}}$ with $\mathsf{Open}(\tau_i^e)$. All players get e_i. If any opening failed, the players output the numbers of the respective players, and the protocol aborts.
4. All players compute $\mathsf{chall} \leftarrow 1 + \left(\left(\sum_{i=1}^n e_i\right) \bmod c\right)$.

Opening Phase:

5. Every player P_i calls $\mathcal{F}_{\text{Commit}}$ with $\mathsf{Open}(\tau_{i,j}^s)$ for all $j \neq \mathsf{chall}$ so that all players obtain the value $s_{i,j}$ for $j \neq \mathsf{chall}$. If any opening fails, the players output the numbers of the respective players, and the protocol aborts.
6. For all $j \neq \mathsf{chall}$ and all $i' \leq n$, the players check whether $\mathfrak{c}_{i',j}$ was generated correctly using $s_{i',j}$. If not, they output the numbers of the respective players i', and the protocol aborts.
7. Otherwise, every player P_i stores $\{\mathfrak{c}_{i',\mathsf{chall}}\}_{i' \leq n}$ and $m_{i,\mathsf{chall}}$.

Fig. 7. Protocol that allows ciphertext to be used as commitments for plaintexts

C.2 Functionalities

C.3 Proof of Theorem 3

Proof. We construct a simulator $\mathcal{S}_{\text{SHE}}$ (see Figure 9) working on top of $\mathcal{F}_{\text{SHE}}$ such that the environment can not distinguish whether it is playing with the real protocol $\Pi_{\text{EncCommit}}$ and $\mathcal{F}_{\text{KeyGen}}$ or with $\mathcal{F}_{\text{SHE}}$ and $\mathcal{S}_{\text{SHE}}$. The simulator is given in Figure 9.

Calls to $\mathcal{F}_{\text{KeyGen}}$ are simulated as in $\mathcal{S}_{\text{KeyGen}}$. We now focus on the commitment phase.

Let A be the set of indices of corrupted players. The simulator starts assuming that the adversary will behave honestly. It samples a uniform $j_0 \leftarrow \{1, \ldots, c\}$ and seeds $\{s_{i,j}\}_{i \notin A, j \neq j_0}$. If the adversary does not deviate, then round j_0 will remain unopened, otherwise the simulator will have to adjust this. We can simulate each round j as follows. First, the simulator gets corrupted seeds $s_{i,j}$ for $i \in A$ when the adversary commits to them in step 2a. It gives in return random handles $\tau_{i,j}^s$ on behalf of each honest player P_i.

If $j \neq j_0$, the simulator engages with the adversary in a normal run of steps 2b to 2c using seeds $s_{i,j}$ for honest player P_i. Since the simulator knows the corrupt seeds of the current round j, it can check whether the adversary behaved honestly. If the adversary did not, then the simulator stores index j in the cheating list.

If $j = j_0$, the simulator checks again whether the adversary computed the right encryptions $\{\mathfrak{c}_i\}_{i \in A}$. If it did not, the simulator stores j_0 in the cheating list. Then the simulator calls $\mathsf{EncCommit}$ to $\mathcal{F}_{\text{SHE}}$ on seeds $\{s_{i,j_0}\}_{i \in A}$ and gets back $\{\mathfrak{c}_i\}_{i \notin A}$, which are the values computed by the functionality. It then sets $\mathfrak{c}_{i,j_0} \leftarrow \mathfrak{c}_i$ and pass them onto the adversary in step 2c.

Once the last round is finished, the simulator checks the cheating list. There are three possibilities:

The ideal functionality $\mathcal{F}_{\text{SHE}}$

Usage: The functionality is split into a one-run stage which computes the key material and a stage which can be accessed several times and is targeted to replace the zero-knowledge protocols in [14].

KeyGen: On input KeyGen the functionality acts as a copy of $\mathcal{F}_{\text{KEYGEN}}$.
Notice that all the variables used during this call are available for later use.

EncCommit: On input EncCommit the functionality does the following.

Initialize: Denote by A the set of indices of corrupt players. On input Start by all players, sample, at random, seeds $\{s_i\}_{i\notin A}$ and wait for corrupted seeds $\{s_i\}_{i\in A}$ from the adversary.

Computation:

1. It sets $\mathbf{m}_i \leftarrow \mathsf{PRF}_{s_i}$ subject to condition cond.
2. It sets $\mathfrak{c}_i = \mathsf{Enc}_{\mathfrak{pk}}(\mathbf{m}_i, \mathcal{RC}_{s_i}(0.5, \sigma^2, \phi(m)))$ for each player P_i.
3. It gives $\{\mathfrak{c}_i\}_{i\notin A}$ to the adversary, and waits for signal Deliver, Cheat or Abort.

Delivery: The functionality sends $\mathbf{m}_i, \{\mathfrak{c}_j\}_{j\leq n}$ to player P_i.

Cheat: The functionality gives $\{s_i\}_{i\notin A}$ to the adversary, then it decides to do either of the following things:

- With probability $1/c$ it sends Success to the adversary, it waits for $\{\mathbf{m}_i, \mathfrak{c}_i\}_{i\in A}$, and outputs $\mathbf{m}_i, \{\mathfrak{c}_j\}_{i\leq n}$ to player P_i.
- Otherwise sends NoSuccess to the adversary, and goes to abort.

Abort: The functionality waits for the adversary to input $S \subseteq A$, and outputs S to all players.

Fig. 8. The ideal functionality for key generation and $\Pi_{\text{ENCCOMMIT}}$.

- The list is empty. In other words, the adversary behaved honestly. The simulator sets $\mathsf{chall} \leftarrow j_0$, and sends Deliver to the functionality if all commitments are successfully opened. The output of $\mathcal{F}_{\text{SHE}}$ and what the adversary has already seen seen will be consistent since $\mathcal{F}_{\text{SHE}}$ was called in round j_0 with the right seeds $\{s_{i,j_0}\}_{i\in A}$.
- The list contains only one index j_1. In this case the simulator sends Cheat and gets in return seeds $\{s_i\}_{i\notin A}$ used by the functionality. It sets $s_{i,j_0} \leftarrow s_i$ for each honest player P_i. It then waits for the answer.
  - If the functionality returns Success, the simulator has to make the adversary believe that round j_1 will remain unopened. It sets $\mathsf{chall} \leftarrow j_1$. If all commitments are successfully opened, it sends $\{\mathbf{m}_{i,j_1}, \mathfrak{c}_{i,j_1}\}_{i\in A}$ to the functionality in order to make consistent players' outputs and what the adversary has already seen.
  - If it returns NoSuccess, the simulator has to make the adversary believe that round j_1 will be opened. Therefore it samples $\mathsf{chall} \leftarrow \{1, \ldots, c\} \setminus \{j_1\}$.
- The list contains at least two indices j_1, j_2. In this case the real protocol would result in abort, so the simulator sends Abort to the functionality and sets $\mathsf{chall} \leftarrow j_0$.

Later the simulator generates the value e_i for each honest player such that $1 + ((\sum_{i=1}^{n} e_i) \bmod c) = \mathsf{chall}$. This ensures that once the challenge is computed, it will point to a round in the same fashion as the protocol would do. Moreover, opening τ_i^e to (any) e_i is does not give clues to the adversary if it is playing in a real run of the protocol or in a simulated one.

In the opening phase, the simulator gives $\{e_i\}_{i\notin A}$ and honest share $\{s_{i,j}\}_{i\notin A, j\neq \mathsf{chall}}$ to the adversary, and if it there was a cheating with no success, then it also sends Abort on behalf of each honest player.

It is clear, from the construction of $\mathcal{F}_{\text{SHE}}$, that all the messages generated by the simulator are indistinguishable from a real run of the protocol. The simulator does the same computations, except in round j_0 where the computation is done by the functionality, and the values are then passed onto the simulator, which forwards them to the adversary.

Finally, if the protocol aborts due to failure at opening commitments, both the functionality and the players output the numbers of corrupted players who failed to open their commitments. If the protocol aborts at step 6, the output is the numbers of players who deviated in threads other than chall in both the functionality and the protocol. □

The simulator $\mathcal{S}_{\text{SHE}}$

KeyGen: $\mathcal{S}_{\text{SHE}}$ acts as $\mathcal{S}_{\text{KEYGEN}}$, but $\mathcal{S}_{\text{SHE}}$ calls $\mathcal{F}_{\text{SHE}}$ on query KeyGen, when $\mathcal{S}_{\text{KEYGEN}}$ would have called $\mathcal{F}_{\text{KEYGEN}}$.

Commitment Phase:

- The simulator chooses random $j_0 \leftarrow \{1, \ldots, c\}$ and seeds $\{s_{i,j}\}_{i \notin A, j \neq j_0}$.
- Acting as the $\mathcal{F}_{\text{COMMIT}}$ functionality, in response to query in step 1 and 2a, for $j = 1, \ldots, c$ the simulator samples $s_{i,j}$ according to the protocol for $i \notin A$ and returns random handles $\{\tau_i^e\}_{i \leq n}, \{\tau_{i,j}^s\}_{i \leq n}$.
- For $j = 1, \ldots, c$, the simulator does the following:
  - If $j \neq j_0$, it performs steps 2b and 2c according to protocol using honest seeds $s_{i,j}$ for each $i \notin A$.
  - If $j = j_0$, it calls $\mathcal{F}_{\text{SHE}}$ on query EncCommit on corrupted seeds $\{s_{i,j_0}\}_{i \in A}$ and gets back honest encryptions $\{\mathfrak{c}_i\}_{i \notin A}$. It then sets $\mathfrak{c}_{i,j_0} \leftarrow \mathfrak{c}_i$ for each $i \notin A$.
- In step 2c, the simulator receives encryptions $\mathfrak{c}_{i,j}^*$ for each $i \in A$ and $j \in \{1, \ldots, c\}$. It generates $\mathbf{m}_{i,j}$ subject to cond, and $\mathfrak{c}_{i,j} \leftarrow \mathsf{Enc}_{\mathfrak{pk}}(\mathbf{m}_{i,j})$, and checks if $\mathfrak{c}_{i,j} = \mathfrak{c}_{i,j}^*$. If the equality does not hold, it stores j in a (cheating) list.
- The simulator reads the cheating list. There are three possibilities:
  - The list is empty. The simulator sets chall $\leftarrow j_0$.
  - The list contains only one index j_1. The simulator sends Cheat to $\mathcal{F}_{\text{SHE}}$ and gets $\{s_i\}_{i \notin A}$ back. It then sets $s_{i,j_0} \leftarrow s_i$ for each $i \notin A$.
    - If the functionality returns Success, the simulator sets chall $\leftarrow j_1$.
    - If the functionality returns NoSuccess, the simulator samples chall $\leftarrow \{1, \ldots, c\} \setminus \{j_1\}$.
  - The list contains at least two indices. The simulator sends Abort to $\mathcal{F}_{\text{SHE}}$, gets $\{s_i\}_{i \notin A}$ and sets $s_{i,j_0} \leftarrow s_i$ for each $i \notin A$, and chall $\leftarrow j_0$.
- For all honest P_i the simulator sets e_i uniformly in $1, \ldots, c$ with the constraint $1 + \left(\left(\sum_{i=1}^n e_i\right) \bmod c\right) = \mathsf{chall}$.
- In step 3, the simulator opens the handle τ_i^e to the freshly defined value e_i, for all honest P_i. If the adversary fails to open some of the commitments of corrupted players, the simulator sends Abort and the numbers of the respective players to $\mathcal{F}_{\text{SHE}}$, and it stops.
- Step 4 is performed according to the protocol.

Opening Phase:

- In step 5, the simulator opens the handle $\tau_{i,j}^s$ to $s_{i,j}$ for all honest players $i \notin A$ and $j \neq \mathsf{chall}$. If the adversary fails to open some of the commitments of corrupted players, the simulator sends Abort and the numbers of the respective players to $\mathcal{F}_{\text{SHE}}$, and it stops.
- If the cheating list is empty, the simulator sends Deliver to $\mathcal{F}_{\text{SHE}}$.
- If the functionality returned Success earlier, the simulator inputs $\{\mathbf{m}_{i,\mathsf{chall}}, \mathfrak{c}_{i,\mathsf{chall}}^*\}_{i \in A}$ to the functionality.
- If the functionality returned NoSuccess, or if the cheating list has at least two indices, the simulator inputs to the functionality the number of players $i \in A$ whose $\mathfrak{c}_{i,j}^*$ were computed incorrectly for some $j \neq \mathsf{chall}$.

Fig. 9. The simulator for $\mathcal{F}_{\text{SHE}}$

D Offline Phase : Protocol, Functionalities and Simulators

D.1 Protocols

Protocol MACCheck

Usage: Each player has input α_i and $(\gamma(a_j)_i)$ for $j = 1, \ldots, t$. All players have a public set of opened values $\{a_1, \ldots, a_t\}$; the protocol either succeeds or outputs failure if an inconsistent MAC value is found.

$\mathsf{MACCheck}(\{a_1, \ldots, a_t\})$**:**

1. Every player P_i samples a seed s_i and asks $\mathcal{F}_{\text{COMMIT}}$ to broadcast $\tau_i^s \leftarrow \mathsf{Commit}(s_i)$.
2. Every player P_i calls $\mathcal{F}_{\text{COMMIT}}$ with $\mathsf{Open}(\tau_i^s)$ and all players obtain s_j for all j.
3. Set $s \leftarrow s_1 \oplus \cdots \oplus s_n$.
4. Players sample a random vector $\mathbf{r} = \mathcal{U}_s(p, t)$; note all players obtain the same vector as they have agreed on the seed s.
5. Each player computes the public value $a \leftarrow \sum_{j=1}^{t} r_j \cdot a_j$.
6. Player i computes $\gamma_i \leftarrow \sum_{j=1}^{t} r_j \cdot \gamma(a_j)_i$, and $\sigma_i \leftarrow \gamma_i - \alpha_i \cdot a$.
7. Player i asks $\mathcal{F}_{\text{COMMIT}}$ to broadcast $\tau_i^\sigma \leftarrow \mathsf{Commit}(\sigma_i)$.
8. Every player calls $\mathcal{F}_{\text{COMMIT}}$ with $\mathsf{Open}(\tau_i^\sigma)$, and all players obtain σ_j for all j.
9. If $\sigma_1 + \cdots + \sigma_n \neq 0$, the players output $\varnothing$ and abort.

Fig. 10. Method To Check MACs On Partially Opened Values

Protocol Reshare

Usage: Input is $\mathfrak{c}_{\mathbf{m}}$, where $\mathfrak{c}_{\mathbf{m}} = \mathsf{Enc}_{\mathfrak{pk}}(\mathbf{m})$ is a public ciphertext and a parameter enc, where $enc = \mathsf{NewCiphertext}$ or $enc = \mathsf{NoNewCiphertext}$. Output is a share $\mathbf{m}_i$ of $\mathbf{m}$ to each player P_i; and if $enc = \mathsf{NewCiphertext}$, a ciphertext $\mathfrak{c}'_{\mathbf{m}}$. The idea is that $\mathfrak{c}_{\mathbf{m}}$ could be a product of two ciphertexts, which Reshare converts to a "fresh" ciphertext $\mathfrak{c}'_{\mathbf{m}}$. Since Reshare uses distributed decryption (that may return an incorrect result), it is not guaranteed that $\mathfrak{c}_{\mathbf{m}}$ and $\mathfrak{c}'_{\mathbf{m}}$ contain the same value, but it *is* guaranteed that $\sum_i \mathbf{m}_i$ is the value contained in $\mathfrak{c}'_{\mathbf{m}}$.

$\mathsf{Reshare}(\mathfrak{c}_{\mathbf{m}}, enc)$:

1. The players run $\mathcal{F}_{\text{SHE}}$ on query $\mathsf{EncCommit}(R_p)$ so that player i obtains plaintext $\mathbf{f}_i$ and all players obtain $\mathfrak{c}_{\mathbf{f}_i}$, an encryption of $\mathbf{f}_i$.
2. The players compute $\mathfrak{c}_{\mathbf{f}} \leftarrow \mathfrak{c}_{\mathbf{f}_1} + \cdots + \mathfrak{c}_{\mathbf{f}_n}$, and $\mathfrak{c}_{\mathbf{m}+\mathbf{f}} \leftarrow \mathfrak{c}_{\mathbf{m}} + \mathfrak{c}_{\mathbf{f}}$. We define $\mathbf{f} = \mathbf{f}_1 + \cdots + \mathbf{f}_n$, although no party can compute $\mathbf{f}$.
3. The players invoke Protocol DistDec to decrypt $\mathfrak{c}_{\mathbf{m}+\mathbf{f}}$ and thereby obtain $\mathbf{m} + \mathbf{f}$.
4. P_1 sets $\mathbf{m}_1 \leftarrow \mathbf{m} + \mathbf{f} - \mathbf{f}_1$, and each player P_i ($i \neq 1$) sets $\mathbf{m}_i \leftarrow -\mathbf{f}_i$.
5. If $enc = \mathsf{NewCiphertext}$, all players set $\mathfrak{c}'_{\mathbf{m}} \leftarrow \mathsf{Enc}_{\mathfrak{pk}}(\mathbf{m} + \mathbf{f}) - \mathfrak{c}_{\mathbf{f}_1} - \cdots - \mathfrak{c}_{\mathbf{f}_n}$, where a default value for the randomness is used when computing $\mathsf{Enc}_{\mathfrak{pk}}(\mathbf{m} + \mathbf{f})$.

Fig. 11. The Protocol For Additively Secret Sharing A Plaintext $\mathbf{m} \in R_p$ On Input A Ciphertext $\mathfrak{c}_{\mathbf{m}} = \mathsf{Enc}_{\mathfrak{pk}}(\mathbf{m})$.

Protocol Π_{PREP}

Usage: Note that DataGeneration can be run in four distinct threads, and DataCheck in two threads with one thread executing the Square and Shared bit checking at the same time. Each thread executes its own check for correct broadcasting using Section 3.1.

Initialize: This produces the keys for encryption and MACs. On input (Start, p) from all the players:

1. The players call $\mathcal{F}_{\text{SHE}}$ on query KeyGen so player i obtains $(\mathfrak{s}_i, \mathfrak{pk}, \mathfrak{enc})$.
2. The players call $\mathcal{F}_{\text{SHE}}$ on query $\mathsf{EncCommit}(\mathbb{F}_p)$ so player j obtains a share α_j of the MAC key, and all players get $\mathfrak{c}_i$, and encryption of α_i, for $1 \leq i \leq n$.
3. All players set $\mathfrak{c}_\alpha \leftarrow \mathfrak{c}_1 + \cdots + \mathfrak{c}_n$.

Data Generation: On input $(\mathsf{DataGen}, n_I, n_m, n_s, n_b)$, the players execute the following subprocedures of DataGen from Figure 13 and Figure 14:

1. $\mathsf{InputProduction}(n_I)$
2. $\mathsf{Triples}(n_m)$
3. $\mathsf{Squares}(n_s)$
4. $\mathsf{Bits}(n_b)$

Data Check: On input DataCheck, the players do the following:

1. Generate two random values t_m, t_{sb} running the steps below twice:
   (a) Every player P_i samples random $t_i \leftarrow \mathbb{F}_p$ and asks $\mathcal{F}_{\text{COMMIT}}$ to broadcast $\tau_i^t \leftarrow \mathsf{Commit}(t_i)$.
   (b) Every player P_i calls $\mathcal{F}_{\text{COMMIT}}$ with $\mathsf{Open}(\tau_i^t)$ and all players obtain t_j for $1 \leq j \leq n$.
   (c) Every player sets $t \leftarrow t_1 + \cdots + t_n$. If $t = 0$, then repeat the previous steps.
2. Execute $\mathsf{DataCheck}(t_m, t_{sb})$.

Finalize: For the set of partially opened values run protocol MACCheck from Figure 10.

Abort: If $\mathcal{F}_{\text{SHE}}$ outputs a set S of corrupted players at any time, all players output S, and the protocol aborts.

Fig. 12. The Preprocessing Phase

Procedure DataGen

Input Production: This produces at least $n_I \cdot n$ shared values $r_{i,j}$ for $1 \leq i \leq n_I$ and $1 \leq j \leq n$ such that player j holds the actual value $r_{i,j}$ and all other players hold a sharing of this value only.

1. For $j \in \{1, \ldots, n\}$ and $k \in \{1, \ldots, \lceil 2 \cdot n_I/m \rceil\}$.
   (a) Player j generates $\mathbf{r} \in R_p$.
   (b) Player j computes $\mathfrak{c} \leftarrow \mathsf{Enc}_{\mathfrak{pk}}(\mathbf{r})$ and broadcasts the ciphertext to all players.
   (c) The parties execute $\mathsf{Reshare}(\mathfrak{c}, \mathsf{NoNewCiphertext})$ so that player i obtains the share $\mathbf{r}_i$ of $\mathbf{r}$
   (d) All parties compute $\mathfrak{c}_{\gamma(\mathbf{r})} \leftarrow \mathfrak{c}_{\mathbf{r}} \cdot \mathfrak{c}_{\alpha}$.
   (e) The parties execute $\mathsf{Reshare}(\mathfrak{c}_{\gamma(\mathbf{r})}, \mathsf{NoNewCiphertext})$. to obtain shares $\gamma(\mathbf{r})_i$.
   (f) Player i decomposes the plaintext elements $\mathbf{r}_i$ and $\gamma(\mathbf{r})_i$ into their $m/2$ slot values via the FFT and locally stores the resulting data.
   (g) Player j does the same with $\mathbf{r}$ to obtain the values $r_{(k-1)\cdot m/2+i,j}$ for $i = 1, \ldots, m/2$.

Triples: This produces at least $2 \cdot n_m$ $\langle\cdot\rangle$-shared values (a_j, b_j, c_j) such that $c_j = a_j \cdot b_j$.

1. For $k \in \{1, \ldots, \lceil 4 \cdot n_m/m \rceil\}$.
   (a) The players run $\mathcal{F}_{\text{SHE}}$ on query $\mathsf{EncCommit}(R_p)$ so that player i obtains plaintext $\mathbf{a}_i$ and all players obtain $\mathfrak{c}_{\mathbf{a}_i}$ an encryption of $\mathbf{a}_i$.
   (b) The players compute $\mathfrak{c}_{\mathbf{a}} \leftarrow \mathfrak{c}_{\mathbf{a}_1} + \cdots + \mathfrak{c}_{\mathbf{a}_n}$ We define $\mathbf{a} = \mathbf{a}_1 + \cdots + \mathbf{a}_n$, although no party can compute $\mathbf{a}$.
   (c) The players run $\mathcal{F}_{\text{SHE}}$ on query $\mathsf{EncCommit}(R_p)$ so that player i obtains plaintext $\mathbf{b}_i$ and all players obtain $\mathfrak{c}_{\mathbf{b}_i}$ an encryption of $\mathbf{b}_i$.
   (d) The players compute $\mathfrak{c}_{\mathbf{b}} \leftarrow \mathfrak{c}_{\mathbf{b}_1} + \cdots + \mathfrak{c}_{\mathbf{b}_n}$ We define $\mathbf{b} = \mathbf{b}_1 + \cdots + \mathbf{b}_n$, although no party can compute $\mathbf{b}$.
   (e) All parties compute $\mathfrak{c}_{\mathbf{a}\cdot\mathbf{b}} \leftarrow \mathfrak{c}_{\mathbf{a}} \cdot \mathfrak{c}_{\mathbf{b}}$.
   (f) The parties execute $\mathsf{Reshare}(\mathfrak{c}_{\mathbf{a}\cdot\mathbf{b}}, \mathsf{NewCiphertext})$ so that player i obtains the share $\mathbf{c}_i$ and all players obtain a ciphertext $\mathfrak{c}_{\mathbf{c}}$ encrypting the plaintext $\mathbf{c} = \mathbf{c}_1 + \cdots + \mathbf{c}_n$.
   (g) All parties compute $\mathfrak{c}_{\gamma(\mathbf{a})} \leftarrow \mathfrak{c}_{\mathbf{a}} \cdot \mathfrak{c}_{\alpha}$, $\mathfrak{c}_{\gamma(\mathbf{b})} \leftarrow \mathfrak{c}_{\mathbf{b}} \cdot \mathfrak{c}_{\alpha}$ and $\mathfrak{c}_{\gamma(\mathbf{c})} \leftarrow \mathfrak{c}_{\mathbf{c}} \cdot \mathfrak{c}_{\alpha}$.
   (h) The parties execute $\mathsf{Reshare}(\mathfrak{c}_{\gamma(\mathbf{a})}, \mathsf{NoNewCiphertext})$, $\mathsf{Reshare}(\mathfrak{c}_{\gamma(\mathbf{b})}, \mathsf{NoNewCiphertext})$ and $\mathsf{Reshare}(\mathfrak{c}_{\gamma(\mathbf{c})}, \mathsf{NoNewCiphertext})$ to obtain shares $\gamma(\mathbf{a})_i$, $\gamma(\mathbf{b})_i$ and $\gamma(\mathbf{c})_i$.
   (i) Player i decomposes the various plaintext elements into their $m/2$ slot values via the FFT and locally stores the resulting $m/2$ multiplication triples.

Fig. 13. Production Of Tuples and Shared Bits

Procedure DataGen

Squares: This produces at least $(2 \cdot n_s + n_b)$ $\langle \cdot \rangle$-shared values (a_j, b_j) such that $b_j = a_j \cdot a_j$.

1. For $k \in \{1, \ldots, \lceil 2 \cdot (2 \cdot n_s + n_b)/m \rceil\}$.
   (a) The players run $\mathcal{F}_{\text{SHE}}$ on query $\mathsf{EncCommit}(R_p)$ so that player i obtains plaintext $\mathbf{a}_i$ and all players obtain $\mathfrak{c}_{\mathbf{a}_i}$ an encryption of $\mathbf{a}_i$.
   (b) The players compute $\mathfrak{c}_{\mathbf{a}} \leftarrow \mathfrak{c}_{\mathbf{a}_1} + \cdots + \mathfrak{c}_{\mathbf{a}_n}$ We define $\mathbf{a} = \mathbf{a}_1 + \cdots + \mathbf{a}_n$, although no party can compute $\mathbf{a}$.
   (c) All parties compute $\mathfrak{c}_{\mathbf{a}^2} \leftarrow \mathfrak{c}_{\mathbf{a}} \cdot \mathfrak{c}_{\mathbf{a}}$.
   (d) The parties execute $\mathsf{Reshare}(\mathfrak{c}_{\mathbf{a}^2}, \mathsf{NewCiphertext})$ so that player i obtains the share $\mathbf{b}_i$ and all players obtain a ciphertext $\mathfrak{c}_{\mathbf{b}}$ encrypting the plaintext $\mathbf{b} = \mathbf{b}_1 + \cdots + \mathbf{b}_n$.
   (e) All parties compute $\mathfrak{c}_{\gamma(\mathbf{a})} \leftarrow \mathfrak{c}_{\mathbf{a}} \cdot \mathfrak{c}_{\alpha}$ and $\mathfrak{c}_{\gamma(\mathbf{b})} \leftarrow \mathfrak{c}_{\mathbf{b}} \cdot \mathfrak{c}_{\alpha}$.
   (f) The parties execute $\mathsf{Reshare}(\mathfrak{c}_{\gamma(\mathbf{a})}, \mathsf{NoNewCiphertext})$ and $\mathsf{Reshare}(\mathfrak{c}_{\gamma(\mathbf{b})}, \mathsf{NoNewCiphertext})$ to obtain shares $\gamma(\mathbf{a})_i$ and $\gamma(\mathbf{b})_i$.
   (g) Player i decomposes the various plaintext elements into their $m/2$ slot values via the FFT and locally stores the resulting $m/2$ squaring tuples.

Bits: This produces at least n_b $\langle \cdot \rangle$-shared values b_j such that $b_j \in \{0, 1\}$.

1. For $k \in \{1, \ldots, \lceil 2 \cdot n_b/m \rceil + 1\}$.[a]
   (a) The players run $\mathcal{F}_{\text{SHE}}$ on query $\mathsf{EncCommit}(R_p)$ so that player i obtains plaintext $\mathbf{a}_i$ and all players obtain $\mathfrak{c}_{\mathbf{a}_i}$ an encryption of $\mathbf{a}_i$.
   (b) The players compute $\mathfrak{c}_{\mathbf{a}} \leftarrow \mathfrak{c}_{\mathbf{a}_1} + \cdots + \mathfrak{c}_{\mathbf{a}_n}$ We define $\mathbf{a} = \mathbf{a}_1 + \cdots + \mathbf{a}_n$, although no party can compute $\mathbf{a}$.
   (c) All parties compute $\mathfrak{c}_{\mathbf{a}^2} \leftarrow \mathfrak{c}_{\mathbf{a}} \cdot \mathfrak{c}_{\mathbf{a}}$.
   (d) The players invoke protocol $\mathsf{DistDec}$ to decrypt $\mathfrak{c}_{\mathbf{a}^2}$ and thereby obtain $\mathbf{s} = \mathbf{a}^2$.
   (e) If any slot position in $\mathbf{s}$ is equal to zero then set it to one. .
   (f) A fixed square root $\mathbf{t}$ of $\mathbf{s}$ is taken, say the one for which each slot position is odd when represented in $[1, \ldots, p)$.
   (g) Compute $\mathfrak{c}_{\mathbf{v}} \leftarrow \mathbf{t}^{-1} \cdot \mathfrak{c}_{\mathbf{a}}$, this is an encryption of $\mathbf{v} = \mathbf{t}^{-1} \cdot \mathbf{a}$, which is a message for which each slot position contains $\{-1, 1\}$, bar the one which we replaced in step (1e).
   (h) All parties compute $\mathfrak{c}_{\gamma(\mathbf{v})} \leftarrow \mathfrak{c}_{\mathbf{v}} \cdot \mathfrak{c}_{\alpha}$.
   (i) The parties execute $\mathsf{Reshare}(\mathfrak{c}_{\mathbf{v}}, \mathsf{NoNewCiphertext})$ and $\mathsf{Reshare}(\mathfrak{c}_{\gamma(\mathbf{v})}, \mathsf{NoNewCiphertext})$ to obtain shares $\mathbf{v}_i$ and $\gamma(\mathbf{v})_i$.
   (j) Player i decomposes the various plaintext elements into their slot values via the FFT, bar the ones replaced in step (1e) to obtain $\langle v_j \rangle$ for $j = 1, \ldots, B$ where $B \approx m \cdot (p-1)/(2 \cdot p)$.
   (k) Set $\langle b_j \rangle \leftarrow (1/2) \cdot (\langle v_j \rangle + 1)$ and output $\langle b_j \rangle$.

[a] Notice that in the production of shared bits the number of rounds is one more than one would expect at first glance: this is because some entry of the input vector may be equal to zero, making such entry unusable for the procedure. This event happens with probability $1/p$, so the expected number of bits produced per iteration is $m \cdot (p-1)/(2 \cdot p)$, rather than $m/2$ (if no entry were zero). Therefore, in order to produce at least n_b elements, we add an extra round to the procedure.

Fig. 14. Production Of Tuples and Shared Bits (continued)

Procedure DataCheck

Usage: Note that all players have previously agreed on two common random values t_m, t_{sb}.

Checking Multiplication Triples: This produces at least n_m checked $\langle\cdot\rangle$-shared values (a_j, b_j, c_j) such that $c_j = a_j \cdot b_j$.

1. For $k \in \{1, \ldots, n_m\}$.
   (a) Take two unused multiplication tuples $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$, $(\langle f\rangle, \langle g\rangle, \langle h\rangle)$ from the list determined earlier.
   (b) Partially open $t_m \cdot \langle a\rangle - \langle f\rangle$ to obtain ρ and $\langle b\rangle - \langle g\rangle$ to obtain σ.
   (c) Evaluate $t_m \cdot \langle c\rangle - \langle h\rangle - \sigma \cdot \langle f\rangle - \rho \cdot \langle g\rangle - \sigma \cdot \rho$ and partially open the result to obtain τ.
   (d) If $\tau \neq 0$ then output $\varnothing$ and abort.
   (e) Output $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ as a valid multiplication triple.

Checking Squaring Tuples: This produces at least n_s checked $\langle\cdot\rangle$-shared values (a_j, b_j) such that $b_j = a_j^2$.

1. For $k \in \{1, \ldots, n_s\}$.
   (a) Take two unused squaring tuples $(\langle a\rangle, \langle b\rangle)$, $(\langle f\rangle, \langle h\rangle)$ from the list determined earlier.
   (b) Partially open $t_{sb} \cdot \langle a\rangle - \langle f\rangle$ to obtain ρ.
   (c) Evaluate $t_{sb}^2 \cdot \langle b\rangle - \langle h\rangle - \rho \cdot (t_{sb} \cdot \langle a\rangle + \langle f\rangle)$ and partially open the result to obtain τ.
   (d) If $\tau \neq 0$ then output $\varnothing$ and abort.
   (e) Output $(\langle a\rangle, \langle b\rangle)$ as a valid squaring tuple.

Checking Shared Bits: This produces at least n_b checked $\langle\cdot\rangle$-shared values b_j such that $b_j \in \{0, 1\}$.

1. For $k \in \{1, \ldots, n_b\}$.
   (a) Take an unused squaring tuples $(\langle f\rangle, \langle h\rangle)$ and an unused bit sharing $\langle a\rangle$ from the lists determined earlier.
   (b) Partially open $t_{sb} \cdot \langle a\rangle - \langle f\rangle$ to obtain ρ.
   (c) Evaluate $t_{sb}^2 \cdot \langle a\rangle - \langle h\rangle - \rho \cdot (t_{sb} \cdot \langle a\rangle + \langle f\rangle)$ and partially open the result to obtain τ.
   (d) If $\tau \neq 0$ then output $\varnothing$ and abort.
   (e) Output $\langle a\rangle$ as a valid bit sharing.

Fig. 15. Check The Output Of The Data Production Procedure

D.2 Functionalities

The functionality $\mathcal{F}_{\text{PREP}}$

Let A be the set of indices of corrupted players. Symbols in bold denote vectors in $(\mathbb{F}_p)^k$. Arithmetic is componentwise.

Initialize: On input (Start, p) from honest players and adversary, the functionality sets the internal flag BreakDown to false and then it does the following:

1. For each corrupted player $i \in A$, the functionality accepts shares α_i from the adversary, and it samples at random α_i for each $i \notin A$. Then the functionality sets $\alpha \leftarrow \alpha_1 + \cdots + \alpha_n$.
2. The functionality waits for signal Abort, Proceed or Cheat from the adversary.
3. If received Proceed, the functionality outputs α_i to player i.
4. Otherwise, and if the functionality did not abort in Cheat, it outputs adversary's contribution Δ_i to player i.

Computation: On input DataGen from all honest players and adversary, and only if the functionality received Proceed (or BreakDown is true) it executes the data generation procedures specified in Figure 17.

Macro $\mathsf{Angle}(\mathbf{v}_1, \ldots, \mathbf{v}_n, \Delta_\gamma, k)$ The above will be run by the functionality at several points to create representations $\langle\cdot\rangle$.

1. It gets $\{\gamma_i\}_{i \in A}$ from the adversary.
2. Let $\mathbf{v} = \mathbf{v}_1 + \cdots + \mathbf{v}_n$, set $\gamma(\mathbf{v}) \leftarrow \alpha \cdot \mathbf{v} + \Delta_\gamma$.
3. Sample at random $\gamma_i(\mathbf{v}) \leftarrow (\mathbb{F}_p)^k$ for $i \notin A$, subject to $\gamma(\mathbf{v}) = \sum_i^n \gamma(\mathbf{v})_i$.
4. Return $(\gamma(\mathbf{v})_1, \ldots, \gamma(\mathbf{v})_n)$.

Cheat: The functionality chooses to do either one of the following:

- It sends, with probability $1/c$, Success to the adversary and sets the internal flag BreakDown to true.
- Otherwise it sends NoSuccess to the adversary and players, and goes to "Abort".

Abort: The functionality waits for $S \subseteq A$ from the adversary and then outputs S to all players.

Fig. 16. MAC Generation and Covert Procedures to Generate Auxiliar Data

The functionality $\mathcal{F}_{\text{PREP}}$ (continued)

Let A be the set of indices of corrupted players. Symbols in bold denote vectors in $(\mathbb{F}_p)^k$. Arithmetic is componentwise.

Input Production: On input $\mathsf{DataType} = (\mathsf{InputPrep}, n_I)$,

1. The functionality choose random values $I = \{\mathbf{r}^{(i)} \in (\mathbb{F}_p)^{n_I} \mid i \notin A\}$.
2. It accepts from the adversary corrupted values $\{\mathbf{r}^{(i)} \in (\mathbb{F}_p)^{n_I} \mid i \in A\}$, corrupted shares $\{\mathbf{r}_k^{(i)} \in (\mathbb{F}_p)^{n_I} \mid k \in A, i \leq n\}$, and offset for data and MACs $\{\Delta_r^{(i)}, \Delta_\gamma^{(i)} \in (\mathbb{F}_p)^{n_I} \mid i \leq n\}$. Then it does the following:
   (a) Sample honest shares $\{\mathbf{r}_k^{(i)} \mid k \notin A, i \leq n\}$ subject to $\mathbf{r}^{(i)} + \Delta_r^{(i)} = \sum_{k=1}^n \mathbf{r}_k^{(i)}$.
   (b) Run macro $\mathsf{Angle}(\mathbf{r}_1^{(i)}, \ldots, \mathbf{r}_n^{(i)}, \Delta_\gamma^{(i)}, n_I)$, for $i \leq n$.
   (c) Output $\{\mathbf{r}^{(i)}, (\mathbf{r}_i^{(j)}, \gamma_i(\mathbf{r}^{(j)}))_{j \leq n}\}$ to player i, or if $\mathsf{BreakDown}$ is true, output adversary's contribution Δ_i to player i.

Multiplication Triples: On input $\mathsf{DataType} = (\mathsf{Triples}, n_m)$,

1. Choose $2 \cdot n_m$ honest shares $I = \{(\mathbf{a}_i, \mathbf{b}_i) \in (\mathbb{F}_p)^{2 \cdot n_m} \mid i \notin A\}$.
2. It accepts corrupted shares $\{(\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i) \in (\mathbb{F}_p)^{3 \cdot n_m} \mid i \in A\}$ and MAC offsets $\{(\Delta_\gamma^{(a)}, \Delta_\gamma^{(b)}, \Delta_\gamma^{(c)}) \in (\mathbb{F}_p)^{3 \cdot n_m}\}$ from the adversary. It performs the following:
   (a) Set $\mathbf{c} \leftarrow (\mathbf{a}_1 + \cdots + \mathbf{a}_n) \cdot (\mathbf{b}_1 + \cdots + \mathbf{b}_n)$.
   (b) Compute a set of honest shares $\{\mathbf{c}_i \mid i \notin A\}$ subject to $\mathbf{c} = \sum_{i=1}^n \mathbf{c}_i$.
   (c) Run the macros $\mathsf{Angle}(\mathbf{a}_1, \ldots, \mathbf{a}_n, \Delta_\gamma^{(a)}, n_m)$, $\mathsf{Angle}(\mathbf{b}_1, \ldots, \mathbf{b}_n, \Delta_\gamma^{(b)}, n_m)$, $\mathsf{Angle}(\mathbf{c}_1, \ldots, \mathbf{c}_n, \Delta_\gamma^{(c)}, n_m)$.
   (d) Output $\{(\mathbf{a}_i, \gamma_i(\mathbf{a})), (\mathbf{b}_i, \gamma_i(\mathbf{b})), (\mathbf{c}_i, \gamma_i(\mathbf{c}))\}$ to player i, or if $\mathsf{BreakDown}$ is true, output adversary's contribution Δ_i to player i.

Squaring Tuples: On input $\mathsf{DataType} = (\mathsf{Squares}, n_s)$,

1. Choose $N = n_s$ honest shares $I = \{\mathbf{a}_i \in (\mathbb{F}_p)^{n_s} \mid i \notin A\}$.
2. It accepts corrupted shares $\{(\mathbf{a}_i, \mathbf{s}_i) \in (\mathbb{F}_p)^{2 \cdot n_s} \mid i \in A\}$ and MAC offsets $\{(\Delta_\gamma^{(a)}, \Delta_\gamma^{(s)}) \in (\mathbb{F}_p)^{2 \cdot n_s}\}$ from the adversary. It does the following:
   (a) Set $\mathbf{s} \leftarrow (\mathbf{a}_1 + \cdots + \mathbf{a}_n) \cdot (\mathbf{a}_1 + \cdots + \mathbf{a}_n)$.
   (b) Compute a a set of honest shares $\{\mathbf{s}_i \mid i \notin A\}$ subject to $\mathbf{s} = \sum_{i=1}^n \mathbf{s}_i$.
   (c) Run the macro $\mathsf{Angle}(\mathbf{a}_1, \ldots, \mathbf{a}_n, \Delta_\gamma^{(a)}, n_s)$ and $\mathsf{Angle}(\mathbf{s}_1, \ldots, \mathbf{s}_n, \Delta_\gamma^{(s)}, n_s)$.
   (d) Output $\{(\mathbf{a}_i, \gamma_i(\mathbf{a})), (\mathbf{s}_i, \gamma_i(\mathbf{s}))\}$ to player i, or if $\mathsf{BreakDown}$ is true, output adversary's contribution Δ_i to player i.

Shared Bits: On input $\mathsf{DataType} = (\mathsf{Bits}, n_b)$,

1. It gets shares $\{\mathbf{b}_i \in (\mathbb{F}_p)^{n_b} \mid i \in A\}$ and MAC offsets $\{\Delta_\gamma^{(b)} \in (\mathbb{F}_p)^{n_b}\}$ from the adversary.
   (a) Uniformly sample n_b honest shares $I = \{\mathbf{b}_i \in (\mathbb{F}_p)^{n_b} \mid i \notin A\}$ subject to the condition $\sum_i \mathbf{b}_i \in \{0,1\}^{n_b}$.
   (b) Run the macro $\mathsf{Angle}(\mathbf{b}_1, \ldots, \mathbf{b}_n, \Delta_\gamma^{(b)}, n_b)$.
   (c) Output $(\mathbf{b}_i, \gamma_i(\mathbf{b}))$ to player i, or if $\mathsf{BreakDown}$ is true, output adversary's contribution Δ_i to player i.

Fig. 17. Operations to Generate Auxiliar Data for the Online Phase

D.3 Proof of Lemma 1

Proof.

We here inspect the correctness and the soundness error of the MACCheck protocol. In order to understand the probability of an adversary being able to cheat, we design the following security game.

1. The challenger generates the secret key $\alpha \leftarrow \alpha_1 + \cdots + \alpha_n$ and MACs $\gamma(a_j)_i \leftarrow \alpha \cdot a_j$ and sends messages $a_1, \ldots, a_t$ to the adversary.
2. The adversary sends back messages $a'_1, \ldots, a'_t$.
3. The challenger generates random values $r_1, \ldots, r_t \leftarrow \mathbb{F}_p$ and sends them to the adversary.
4. The adversary provides an error Δ.
5. Set $a \leftarrow \sum_{j=0}^{t} r_j a'_j$, $\gamma_i \leftarrow \sum_{j=0}^{t} r_j \gamma(a_j)_i$, and $\sigma_i \leftarrow \gamma_i - \alpha_i \cdot a$. Now, the challenger checks that $\sigma_1 + \cdots + \sigma_n = \Delta$

The adversary wins the game if there is an i for which $a'_i \neq a_i$ and the final check goes through.

The second step in the game where the adversary sends the a'_i's models the fact that corrupted players can choose to lie about their shares of values opened during the protocol execution. Δ models the fact that the adversary is allowed to introduce errors on the macs.

Now, let us look at the probability of winning the game if the r_i's are randomly chosen. If the check goes through, we have that the following equalities hold:

$$\begin{aligned}
\Delta &= \sum_{i=1}^{n} \sigma_i = \sum_{i=1}^{n} (\gamma_i - \alpha_i \cdot a) \\
&= \sum_{i=1}^{n} \left(\sum_{j=1}^{t} r_j \cdot \gamma(a_j)_i - \alpha_i \cdot \sum_{j=1}^{t} r_j \cdot a'_j \right) \\
&= \sum_{i=1}^{n} \left(\sum_{j=1}^{t} \left(r_j \cdot \gamma(a_j)_i - \alpha_i \cdot r_j \cdot a'_j \right) \right) \\
&= \sum_{j=1}^{t} \left(r_j \cdot \sum_{i=1}^{n} \left(\gamma(a_j)_i - \alpha_i \cdot a'_j \right) \right) \\
&= \sum_{j=1}^{t} r_j \cdot \left(\alpha \cdot a_j - \alpha \cdot a'_j \right) \\
&= \alpha \cdot \sum_{j=1}^{t} r_j \cdot \left(a_j - a'_j \right)
\end{aligned}$$

So, the following equality holds:

$$\alpha \cdot \sum_{j=0}^{t} r_j (a'_j - a_j) = \Delta. \tag{1}$$

First we consider the case where $\sum_{j=0}^{t} r_j (a'_j - a_j) \neq 0$, so $\alpha = \Delta / \sum_{j=0}^{t} r_j (a'_j - a_j)$. This implies that being able to pass the check is equivalent to guessing α. However, since the adversary has no information about α, this happens with probability only $1/|\mathbb{F}_p|$. So what is left is to argue that $\sum_{j=0}^{t} r_j (a'_j - a_j) = 0$ also happens with very low probability. This can be seen as follows. We define $\mu_j := (a'_j - a_j)$ and $\mu := (\mu_1, \ldots, \mu_t), r := (r_1, \ldots, r_t)$. Now $f_\mu(r) := r \cdot \mu = \sum_{j=0}^{t} r_j \mu_j$ defines a linear mapping, which is not the 0-mapping since at least one $\mu_j \neq 0$. From linear algebra we then have the rank-nullity theorem telling us that $\dim(\ker(f_\mu)) = t - 1$. Also since r is random and the adversary does not know r when choosing the a'_i's, the probability of $r \in \ker(f_\mu)$ is $|\mathbb{F}_p^{t-1}|/|\mathbb{F}_p^t| = 1/|\mathbb{F}_p|$. Summing up, the total probability of winning the game is at most $2/|\mathbb{F}_p|$.

For correctness we use the fact that Equation 1 holds with probability one if $a'_j = a_j$ and $\Delta = 0$ (honest prover).

□

D.4 Proof of Theorem 4

Proof. We construct a simulator $\mathcal{S}_{\text{PREP}}$ (given in Figure 18 and Figure 19) such that no polynomial-time environment can distinguish, with significant probability, a view obtained running Π_{PREP} from a view obtained running $\mathcal{S}_{\text{PREP}} \diamond \mathcal{F}_{\text{PREP}}$. The environment's view is the collection of all intermediate messages that corrupted players send and receive, plus the inputs and outputs of all players.

In a nutshell, the simulator will run a copy of Π_{PREP} with the adversary, acting on behalf of honest players. Keys for the underlying cryptosystem and MACs are generated by simulating queries KeyGen and EncCommit to $\mathcal{F}_{\text{SHE}}$ respectively. Note that due to the distributed decryption, data for the (online) input preparation stage might be incorrectly secret shared, and all type of data might be incorrectly MAC'd. Since the simulator knows α and $\mathfrak{s}$, it can compute offsets on the secret sharing and MACs and pass them to $\mathcal{F}_{\text{PREP}}$.

Before we discuss indistinguishability we explain how the cheat mechanism is handled in the simulation. In the execution of Π_{PREP}, the environment may send Cheat either in the initial query KeyGen or in any later query EncCommit to $\mathcal{F}_{\text{SHE}}$. Thus, the success probability depends on the number of cheat attempts. The simulator ensures two things: 1) Whenever the environment sends the *first* Cheat to what it thinks is $\mathcal{F}_{\text{SHE}}$, the call is forwarded to $\mathcal{F}_{\text{PREP}}$, which decides whether or not it is successful. 2) Assuming this cheat was successful, the simulator recreates the success probability that a real interaction would have. This is needed as otherwise the environment would distinguish. The inner procedure SEncCommit is designed for this purpose.

We now turn to show indistinguishability. We point out that there is mainly one difference between a simulated run and a real execution of Π_{PREP}: In a simulated run, honest shares used in the interaction are randomly sampled by the simulator. These shares correspond to the MAC key, and shares of generated data together with the shares of their MACs. At the end of the day, $\mathcal{F}_{\text{PREP}}$ will output data using its own honest shares of α, and its own honest shares of data and MACs.

We can split the view of the environment in four chunks. Namely, messages interchanged either in DataGen, in DataCheck, or in MACCheck, and players' output of $\mathcal{F}_{\text{PREP}}$. Clearly, indistinguishability of simulated and real views of DataGen chunk comes from the semantic property of the underlying cryptosystem. For the DataCheck chunk, note that all opened values are a combination of output data and sacrificed data. The latter does not form part of the final output, and therefore by no means the environment can reconstruct the set of opened values using its view, as it does not know honest shares of the sacrificed data. In other words, openings are randomized via sacrificings from the environment's point of view, so the best it can do is to guess sacrificed honest shares, which happens with probability $1/|\mathbb{F}_p|$ for each share's guessing. For the MACCheck chunk, we refer to the fact that the soundness error of MACCheck is $2/p$, as shown in Lemma 1. Both probabilities are negligible if p is exponential in the security parameter. Lastly, we also have consistency between the output of $\mathcal{F}_{\text{PREP}}$ and what the environment sees in corrupted transcripts. This is due to the fact that the offsets (those quantities denoted by Δ) are simply the difference between deviated and correctly computed data, and therefore independent of what data refers to.

If the protocol aborts in DataCheck or MACCheck, the players output $\varnothing$, and so does $\mathcal{F}_{\text{PREP}}$ on instruction of the simulator. This corresponds to the fact that those protocols do not reveal the identity of any corrupted party.

It remains to show what happens in case Cheat or Abort is sent by the environment. If the cheat did not go through, players' output is a single message S for a set S of corrupted players in both real and simulated interaction. On the other hand, if the cheat did go through, the functionality $\mathcal{F}_{\text{PREP}}$ breaks down, and the simulator can decide what MAC key is used and what data is outputted to every player, so it just gives to $\mathcal{F}_{\text{PREP}}$ what it has been generated during the interaction. If the environment sends Abort and a set S of corrupted players, this is simply passed to $\mathcal{F}_{\text{PREP}}$, which forwards it to the players.

□

The simulator $\mathcal{S}_{\text{PREP}}$

Initialize:

- The simulator first sends (Start, p) to $\mathcal{F}_{\text{PREP}}$ and then interacts with the adversary acting as $\mathcal{F}_{\text{SHE}}$ on query KeyGen to generate the encryption public key $(\mathfrak{pk}, \mathfrak{enc})$ and a complete set of shares $\{\mathfrak{s}_1, \ldots, \mathfrak{s}_n\}$ of the secret key. If the adversary sends Cheat to $\mathcal{F}_{\text{SHE}}$, the simulator forwards it to $\mathcal{F}_{\text{PREP}}$. If the cheat passed through, the simulator sets the flag BreakDown to true, otherwise it is set to false.
- The generation of the MAC key α is done as in the protocol, but calling to $\mathsf{SEncCommit}(\mathbb{F}_p)$ instead to $\mathcal{F}_{\text{SHE}}$ on query EncCommit. The simulator stores $\alpha \leftarrow \alpha_1 + \cdots + \alpha_n$ for later use.
- Lastly, it gives α_i to $\mathcal{F}_{\text{PREP}}$ for $i \in A$ if BreakDown is false, and $i \leq n$ otherwise.
- If the simulation $\mathcal{F}_{\text{SHE}}$ aborts on KeyGen or and EncCommit, go to "Abort".

Command = DataGen: On input (n_I, n_m, n_s, n_b) from honest players and adversary, the simulator sets

$\mathcal{T}_{Input} \leftarrow \mathsf{SimDataGen}(\mathsf{InputPrep}, n_I)$
$\mathcal{T}_{Triples} \leftarrow \mathsf{SimDataGen}(\mathsf{Triples}, n_m)$
$\mathcal{T}_{Squares} \leftarrow \mathsf{SimDataGen}(\mathsf{Squares}, n_s)$
$\mathcal{T}_{Bits} \leftarrow \mathsf{SimDataGen}(\mathsf{Bits}, n_b)$,

where SimDataGen is specified in Figure 19. These calls also return a decision bit. If it is set to Abort, the simulator goes to "Abort".

Command = DataCheck:

- Step 1 is executed as in the protocol but calling to $\mathsf{SEncCommit}(R_p)$. The simulator goes to "Abort" if SEncCommit says so.
- The simulator performs steps (a)-(d) of subprocedures Triples, Squares, Bits of DataCheck. In each iteration k, it gets to know the value σ_k. If any of these values are non-zero, the simulator sends Abort and $\varnothing$ to $\mathcal{F}_{\text{PREP}}$. Otherwise, the algebraic relation among generated data is correct with probability $1 - 1/p$.

Finalize: At this point, the functionality is waiting for instruction Proceed or Abort, or otherwise, a complete break down occurred, and the functionality is waiting for command DataGen and output values from the adversary.

1. The simulator engages with the adversary in a normal run of MACCheck on behalf of each honest player i. Note that to generate honest σ_i the simulator uses shares α_i. If $\sigma_1 + \cdots + \sigma_n \neq 0$, send Abort and $\varnothing$ to $\mathcal{F}_{\text{PREP}}$.
2. Otherwise send Success to the adversary, and send to $\mathcal{F}_{\text{PREP}}$ the following:
   - If BreakDown is false, send $\mathcal{T}_{Input}$, $\mathcal{T}_{Triples}$, $\mathcal{T}_{Squares}$, $\mathcal{T}_{Bits}$.
   - If BreakDown is true, send all the data (corresponding to honest and corrupted players) generated in the execution of SimDataGen.

Abort: If the simulated $\mathcal{F}_{\text{SHE}}$ aborts outputting a set S of corrupted players, input Abort and S to $\mathcal{F}_{\text{PREP}}$.

Fig. 18. The Simulator $\mathcal{S}_{\text{PREP}}$ For The Preprocessing Phase

The simulator $\mathcal{S}_{\text{PREP}}$

SimDataGen(DataType): This procedure gets ready the data to be inputted to $\mathcal{F}_{\text{PREP}}$.

DataType = InputPrep :

- The simulator engages in a normal run of steps (a)-(g) calling to SReshare instead of Reshare. If, at any point, some of the calls returned Abort, the simulator sets Decision $\leftarrow$ Abort and $\mathcal{T}_{Input} \leftarrow \varnothing$.
- Otherwise all the rounds were successful. The simulator sets Decision $\leftarrow$ Continue. Note that in step (c) (after unpacking all the rounds), the simulator gets players' shares and MAC shares $\{\hat{\mathbf{r}_k}^{(i)}, \gamma_k^{(i)} \in (\mathbb{F}_p)^{2 \cdot n_I} \mid i, k \leq n\}$. Then $\hat{\mathbf{r}}^{(i)} = \sum_k^n \hat{\mathbf{r}}_k^{(i)}$ is the (presumably) input of player i. The simulator has the secret key, so it can get the real input $\mathbf{r}^{(i)}$ from the broadcast ciphertexts (if P_i is corrupted) or from what he generated (if P_i is honest). It computes offsets $\Delta_r^{(i)} \leftarrow \hat{\mathbf{r}}^{(i)} - \mathbf{r}^{(i)}$ and $\Delta_\gamma^{(i)} \leftarrow \sum_k^n \gamma_k^{(i)} - \alpha \cdot \mathbf{r}^{(i)}$
  There are two possibilities:
  - Flag BreakDown is set to false. This means no cheat has occurred, so the simulator prepares corrupt inputs, corrupt shares and MAC shares, and offsets. That is, it sets $\mathcal{T}_{Input} \leftarrow \{\mathbf{r}^{(k)}, \hat{\mathbf{r}}_k^{(i)}, \Delta_r^{(i)}, \Delta_\gamma^{(i)}, \gamma_k^{(i)} \mid k \in A, i \leq n\}$
  - Flag BreakDown is set to true. Then there was at least one successful cheat, and the functionality is waiting for adversary's contributions. The simulator sets $\mathcal{T}_{Input}$ to be the output of each player.

DataType = Triples, Squares, Bits: The simulator engages in a normal run of the subprocedure specified by DataType, but calling to SEncCommit(R_p) and SReshare$(\mathfrak{c}_{\mathbf{m}})$ instead of $\mathcal{F}_{\text{ENCCOMMIT}}$ and Reshare$(\mathfrak{c}_{\mathbf{m}})$. If any of the above macros returned Abort the simulator sets Decision $\leftarrow$ Abort and $\mathcal{T}_{DataType} \leftarrow \varnothing$. In any other case the simulator sets Decision $\leftarrow$ Continue, handles the BreakDown flag as above, and does:

- Triples: Set $\mathcal{T}_{Triples} \leftarrow \{(\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i, \gamma(\mathbf{a})_i, \gamma(\mathbf{b})_i, \gamma(\mathbf{c})_i, \Delta_\gamma^{(a)}, \Delta_\gamma^{(b)}, \Delta_\gamma^{(c)}) \in (\mathbb{F}_p)^{9 \cdot (2 \cdot n_m)} \mid i \leq n\}$. The shares are unpacked in step (i): corrupt shares are given by the adversary, and honest shares are sampled uniformly. MAC shares are produced after executing SReshare to simulate step (h), and the offsets are computed as explained earlier.
- Squares: Set $\mathcal{T}_{Squares} \leftarrow \{(\mathbf{a}_i, \mathbf{b}_i, \gamma(\mathbf{a})_i, \gamma(\mathbf{b})_i, \Delta_\gamma^{(a)}, \Delta_\gamma^{(b)}) \in (\mathbb{F}_p)^{6 \cdot (2 \cdot n_s + n_b)} \mid i \leq n\}$ Shares, MAC shares and offsets are obtained as explained above.
- Bits: Set $\mathcal{T}_{Bits} \leftarrow \{(\mathbf{b}_i, \gamma_i, \Delta_\gamma^{(b)}) \in (\mathbb{F}_p)^{3 \cdot (2 \cdot n_b')} \mid i \leq n\}$. A number $n_b' \geq n_b$ of binary shares and MACs has been computed, The exact amount n_b' is round-dependent and it is expected to be approximately $(n_b + m/2) \cdot (p-1)/p$.

Return (Decision, $\mathcal{T}_{DataType}$).

Macro SEncCommit(cond) This macro is intended to simulate a call to $\mathcal{F}_{\text{SHE}}$ on query EncCommit.

- The simulator receives corrupted seeds s_i from the adversary, when it thinks is interacting with $\mathcal{F}_{\text{SHE}}$, and computes $\mathbf{m}_i$ and $\mathfrak{c}_{\mathbf{m}_i}$ for $i \in A$ which are given to the adversary. Then the simulator generates uniformly $\mathbf{m}_i$ and $\mathfrak{c}_i = \text{Enc}_{\mathfrak{pk}}(\mathbf{m}_i)$ for $i \notin A$, and gives $\mathfrak{c}_i$ to the adversary. It waits for response Proceed, Cheat or Abort.
- If the adversary gives Proceed, the simulator sets Decision $\leftarrow$ Continue, and if the adversary gives Abort, set Decision $\leftarrow$ Abort and also send Abort to $\mathcal{F}_{\text{PREP}}$.
- If the adversary gives (Cheat, $\{\mathbf{m}_i^*, \mathfrak{c}_i^*\}_{i \in A}$), set $\mathbf{m}_i \leftarrow \mathbf{m}_i^*$, $\mathfrak{c}_i \leftarrow \mathfrak{c}_i^*$ for $i \in A$, and do the following:
  1. Check if flag BreakDown is false, if so, send Cheat to $\mathcal{F}_{\text{PREP}}$. Then set BreakDown to true. There are two possibilities:
     (a) The functionality returns Success: set Decision $\leftarrow$ Continue.
     (b) The functionality returns NoSuccess: set Decision $\leftarrow$ Abort.
  2. If BreakDown is set to true, with probability $1/c$ set Decision $\leftarrow$ Continue, or otherwise Decision $\leftarrow$ Abort.
- Return (Decision, $\mathbf{m}_1, \ldots, \mathbf{m}_n, \mathfrak{c}_1, \ldots, \mathfrak{c}_n$).

Macro SReshare($\mathfrak{c}_{\mathbf{m}}$)

- Set $(\mathbf{f}_1, \ldots, \mathbf{f}_n, \mathfrak{c}_1, \ldots, \mathfrak{c}_n) \leftarrow$ SEncCommit(R_p) and $\mathbf{f} \leftarrow \sum_i \mathbf{f}_i$. Set Decision $\leftarrow$ Abort if SEncCommit says so.
- Otherwise, set Decision $\leftarrow$ Continue and run steps 2-5 of Reshare. Note that in step 3 the simulator might get an invalid value $(\mathbf{m} + \mathbf{f})^*$. Set $\mathbf{m}_1 \leftarrow (\mathbf{m} + \mathbf{f})^* - \mathbf{f}_1$ and $\mathbf{m}_i \leftarrow -\mathbf{f}_i$.
- Return shares (Decision, $\mathbf{m}_1, \ldots, \mathbf{m}_n$).

Fig. 19. Internal Procedures Of The Simulator $\mathcal{S}_{\text{PREP}}$

E Online Phase : Protocol, Functionalities and Simulators

E.1 Protocols

Protocol Π_{ONLINE}

Initialize: The parties call $\mathcal{F}_{\text{PREP}}$ to get the shares α_i of the MAC key, a number of multiplication triples $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$, squares $(\langle a\rangle, \langle b\rangle)$, bits $\langle b\rangle$, and mask values $(r_i, \langle r_i\rangle)$ as needed for the circuit being evaluated. If $\mathcal{F}_{\text{PREP}}$ aborts outputting a set S of corrupted players, the players output S and abort. Then the operations specified below are performed according to the circuit.

Input: To share his input x_i, Player i takes an available mask value $(r_i, \langle r_i\rangle)$ and does the following:

1. Broadcast $\epsilon \leftarrow x_i - r_i$.
2. The players compute $\langle x_i\rangle \leftarrow \langle r_i\rangle + \epsilon$.

Add: On input $(\langle x\rangle, \langle y\rangle)$, the players locally compute $\langle x+y\rangle \leftarrow \langle x\rangle + \langle y\rangle$.

Multiply: On input $(\langle x\rangle, \langle y\rangle)$, the players do the following:

1. Take one multiplication triple $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ and open $\langle x\rangle - \langle a\rangle, \langle y\rangle - \langle b\rangle$ to get ϵ, ρ respectively.
2. Locally each player sets $\langle z\rangle \leftarrow \langle c\rangle + \epsilon \cdot \langle b\rangle + \rho \cdot \langle a\rangle + \epsilon \cdot \rho$

Square: On input $\langle x\rangle$ the players do the following:

1. Take a square pair $(\langle a\rangle, \langle b\rangle)$ and partially open $\langle x\rangle - \langle a\rangle$ so all players get ϵ.
2. All players locally compute $\langle z\rangle \leftarrow \langle b\rangle + 2 \cdot \epsilon \cdot \langle x\rangle - \epsilon^2$.

Output: This procedure is entered once the players have finished the circuit evaluation, but still the final output $\langle y\rangle$ has not been opened.

1. The players call the MACCheck protocol on input all opened values so far. If it fails, they output $\varnothing$ and abort. $\varnothing$ represents the fact to the corrupted players remain undetected in this case.
2. The players open $\langle y\rangle$ and call MACCheck on input y to verify its MAC. If the check fails, they output $\varnothing$ and abort, otherwise they accept y as a valid output.

Fig. 20. Operations for Secure Function Evaluation

E.2 Functionalities

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

Initialize: On input $(init, p, k)$ from all parties, the functionality stores $(domain, p, k)$ and waits for an input from the environment. Depending on this, the functionality does the following:

- Proceed It sets BreakDown to false and continues.
- Cheat With probability $1/c$, it sets BreakDown to true, outputs Success to the environment and continues. Otherwise it outputs NoSuccess and proceeds as in Abort.
- Abort It waits for the environment to input a set S of corrupted players, outputs it to the players, and aborts.

Input: On input $(input, P_i, varid, x)$ from P_i and $(input, P_i, varid, ?)$ from all other parties, with $varid$ a fresh identifier, the functionality stores $(varid, x)$. If BreakDown is true, it also outputs x to the environment.

Add: On command $(add, varid_1, varid_2, varid_3)$ from all parties (if $varid_1, varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ and stores $(varid_3, x + y)$.

Multiply: On input $(multiply, varid_1, varid_2, varid_3)$ from all parties (if $varid_1, varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ and stores $(varid_3, x \cdot y)$.

Square: On input $(square, varid_1, varid_2)$ from all parties (if $varid_1$ is present in memory and $varid_2$ is not), the functionality retrieves $(varid_1, x)$, and stores $(varid_2, x^2)$.

Output: On input $(output, varid)$ from all honest parties (if $varid$ is present in memory), the functionality retrieves $(varid, y)$ and outputs it to the environment.

- If BreakDown is false, the functionality waits for an input from the environment. If this input is Deliver then y is output to all players. Otherwise $\varnothing$ is output to all players.
- If BreakDown is true, the functionality waits for y^* from the environment and outputs it to all players.

Fig. 21. The ideal functionality for MPC

E.3 Proof of Theorem 5

Proof.

We construct a simulator $\mathcal{S}_{\text{ONLINE}}$ to work on top of the ideal functionality $\mathcal{F}_{\text{ONLINE}}$, such that the adversary cannot distinguish whether it is playing with the protocol Π_{ONLINE} and $\mathcal{F}_{\text{PREP}}$, or the simulator and $\mathcal{F}_{\text{ONLINE}}$. See Appendix E for the complete description of the simulator.

We now proceed with the analysis of the simulation, by first arguing that all the steps before the output are perfectly simulated and finally we show that the simulated output is statistically close to the one in the protocol.

During initialization, the simulator merely acts as $\mathcal{F}_{\text{PREP}}$ with the difference that the decision about the success of a cheating attempt is made by $\mathcal{F}_{\text{ONLINE}}$. If the cheating was successful, $\mathcal{F}_{\text{ONLINE}}$ will output all honest inputs, and the simulator can determine all outputs. Therefore, the simulation will precisely agree with the protocol. For the rest of the proof, we will assume that there was no cheating attempt.

In the input stage the values broadcast by the honest players are uniform in the protocol as well as in the simulation. Addition does not involve communication, while multiplication and squaring involve partial openings: in the protocol a partial opening reveals uniform values, and the same happens also in a simulated run. Moreover, MACs carry the same distribution in both the protocol and the simulation.

In the output stage of both the real and simulated run if the output y is delivered, the environment sees y and the honest players' shares, which are uniform and compatible with y and its MAC. Moreover, in a simulated run the output y is a correct evaluation of the function on the inputs provided by the players in the input phase. In order to conclude, we need to make sure that the same applies to the real protocol with overwhelming probability. As shown in Lemma 1, the adversary was able to cheat in one MACCheck call with probability $2/p$. Thus, the overall cheating probability is negligible since p is assumed to be exponential in the security parameter. This concludes the proof.

□

Simulator $\mathcal{S}_{\text{ONLINE}}$

Initialize: The simulation of the initialization procedure is performed running a local copy of $\mathcal{F}_{\text{PREP}}$. Notice that all the data given to the adversary is know by the simulator.
If the environment inputs Proceed, Cheat, or Abort to the copy of $\mathcal{F}_{\text{PREP}}$, the simulator does so to $\mathcal{F}_{\text{ONLINE}}$ and forwards the output of $\mathcal{F}_{\text{ONLINE}}$ to the environment. If the output is Success, the simulator sets BreakDown to true and uses the environment's inputs as preprocessed data. If $\mathcal{F}_{\text{ONLINE}}$ outputs NoSuccess of the input was Abort, the simulator waits for input S from the environment, forwards it to $\mathcal{F}_{\text{ONLINE}}$, and aborts.

Input:
- If BreakDown is false, honest input is performed according to the protocol, with a dummy input, for example zero.
- If BreakDown is true, $\mathcal{F}_{\text{ONLINE}}$ outputs the inputs of honest players, which then can be used in the simulation.

For inputs given by a corrupt player P_i, the simulator waits for P_i to broadcast the (possibly incorrect) value ϵ', computes $x_i' \leftarrow r_i + \epsilon'$ and uses x_i' as input to $\mathcal{F}_{\text{ONLINE}}$.

Add/Multiply/Square: These procedures are performed according to the protocol. The simulator also calls the respective procedure to $\mathcal{F}_{\text{ONLINE}}$.

Output: $\mathcal{F}_{\text{ONLINE}}$ outputs y to the simulator.
- If BreakDown is false, the simulator now has to provide the honest players' shares of such a value; it already computed an output value y', using the dummy inputs for the honest players, so it can select a random honest player and modify its share adding $y - y'$ and modify the MAC adding $\alpha(y - y')$, which is possible for the simulator, since it knows α. After that, the simulator is ready to open y according to the protocol. If y passes the check, the simulator sends Deliver to $\mathcal{F}_{\text{ONLINE}}$.
- If BreakDown is true, the simulator inputs the result of the simulation to $\mathcal{F}_{\text{ONLINE}}$.

Fig. 22. Simulator for the Online phase

F Active Security

The following is a sketch of a method for an actively secure version of $\Pi_{\text{EncCommit}}$. More specifically, we assume players have access to an ideal functionality $\mathcal{F}^A_{\text{KeyGen}}$ which generates the key material as $\mathcal{F}_{\text{KeyGen}}$, but it models active security rather than covert security. More concretely, this just means that there is no "cheat option" that the adversary can choose. The purpose of this section is therefore to describe a protocol $\Pi^A_{\text{EncCommit}}$ which securely implements an ideal functionality $\mathcal{F}^A_{\text{SHE}}$ in the $\mathcal{F}^A_{\text{KeyGen}}$-hybrid model, where $\mathcal{F}^A_{\text{SHE}}$ behaves as $\mathcal{F}_{\text{SHE}}$, but, again, models active security.

The protocol is inspired by the protocol from [22] where a particularly efficient variant of the cut-and-choose approach was developed.

Let P_i be the player who is to produce ciphertexts to be verified by the other players. The protocol is parametrized by two natural numbers T, b where b divides T. We will set $t = T/b$. The protocol will produce as output t ciphertexts $\mathfrak{c}_0, \ldots, \mathfrak{c}_{t-1}$.

Each such ciphertext is generated according to the algorithm described earlier, and is therefore created from the public key and four polynomials m, v, e_0 and e_1. To make the notation easier to deal with below, we rename these as f_1, f_2, f_3, f_4. We can then observe that there exist ρ_l, for $l = 1, \ldots, 4$ such that $\|f_l\|_\infty \leq \rho_l$ except with negligible probability. Concretely, we can use $\rho_1 = p/2$, $\rho_2 = 1$ and $\rho_3 = \rho_4 = \rho$ where ρ can be determined by a tail-bound on the gaussian distribution used for generating f_3, f_4.

The player P_i will also create a set of random *reference ciphertexts* $\mathfrak{d}_0, \ldots, \mathfrak{d}_{2T-1}$ that are used to verify that $\mathfrak{c}_0, \ldots, \mathfrak{c}_{t-1}$ are well-formed and that P_i knows what they contain. Each $\mathfrak{d}_j$ is created from 4 polynomials $g_1, \ldots, g_4$ in the same way as above, but the polynomials are created with a different distribution. Namely, they are random subject to $\|g_i\|_\infty \leq 4 \cdot \delta \cdot \rho_i \cdot T \cdot \phi(m)$, where $\delta > 1$ is some constant.

The protocol now proceeds as follows:

1. Below P_i is given some number of attempts to prove that his ciphertexts are correctly formed. The protocol is parametrized by a number M which is the maximal number of allowed attempts. We start by setting a counter $v = 1$.
2. P_i broadcasts the ciphertexts $\mathfrak{c}_0, \ldots, \mathfrak{c}_{t-1}$ and the reference ciphertexts $\mathfrak{d}_0, \ldots, \mathfrak{d}_{2T-1}$ containing plaintexts. These ciphertexts should be generated from seeds $s_0, \ldots, s_{2T-1}$ that are first sent through the random oracle and the output is used to generate the plaintext and randomness for the encryptions.
3. A random index subset of size T is chosen, and P_i must broadcast s_i for $i \in T$. Players check that each opened s_i indeed induces the ciphertext $\mathfrak{d}_i$, and abort if this is not the case.
4. A random permutation π on T items is generated and the unopened ciphertexts are permuted according to π. We renumber the permuted ciphertexts and call them $\mathfrak{d}_0, \ldots, \mathfrak{d}_{T-1}$.
5. Now, for each $\mathfrak{c}_i$, the subset of ciphertexts $\{\mathfrak{d}_{bi+j} |\ j = 0, \ldots, b-1\}$ is used to demonstrate that $\mathfrak{c}_i$ is correctly formed. This is called the block of ciphertexts assigned to $\mathfrak{c}_i$. We do as follows:
   (a) For each i, j do the following: let $f_1, \ldots, f_4$ and $g_1, \ldots, g_4$ be the polynomials used to form $\mathfrak{c}_i$, respectively $\mathfrak{d}_{bi+j}$. Define $z_l = f_l + g_l$, for $l = 1, \ldots, 4$.
   (b) Player P_i checks that $\|z_l\|_\infty \leq 4 \cdot \delta \cdot \rho_l \cdot T \cdot \phi(m) - \rho_l$. If this is the case, he broadcasts z_l, for $l = 1, \ldots, 4$. Otherwise he broadcasts $\perp$.
   (c) In the former case players check that $\|z_l\|_\infty$ is in range for $l = 1, \ldots, 4$ and that the z_l's induce the ciphertext $\mathfrak{c}_i + \mathfrak{d}_{bi+j}$.
   (d) At the end, players verify that for each $\mathfrak{c}_i$, P_i has correctly opened $\mathfrak{c}_i + \mathfrak{d}_{bi+j}$ for all ciphertexts in the block assigned to $\mathfrak{c}_i$.
   (e) If all checks go through, output $\mathfrak{c}_0, \ldots, \mathfrak{c}_{t-1}$ and exit. Else, if $v < M$, increment v and go to step 2. Finally, if $v = M$, the prover has failed to convince us M times, so abort the protocol.

It is possible to adapt the protocol for proving that the plaintexts in $\mathfrak{c}_i$ satisfy certain special properties. For instance, assume we want to ensure that the plaintext polynomial f_1 is a constant polynomial, i.e., only the degree-0 coefficient is non-zero. We do this by generating the reference ciphertexts such that for each $\mathfrak{d}_i$, the polynomial g_1 is also a constant polynomial. When opening we check that the plaintext polynomial is always constant. The proof of security is trivially adapted to this case.

Some intuition for why this works: after half the reference ciphertexts are opened, we know that except with exponentially small probability almost all the unopened ciphertexts are well formed. A simulator will be able to extract randomness and plaintext for all the well formed ones. When we split the unopened $\mathfrak{d}_j$'s randomly in blocks of b ciphertexts, it is therefore very unlikely that some block contains only bad ciphertexts. It can be shown that the probability that this happens is at most $t^{1-b} \cdot (e \cdot \ln(2))^{-b}$ [22].

Assume P_i is corrupt: Now, if he survives one iteration of the test, and no block was completely bad, it follows that for every $\mathfrak{c}_i$, he has opened opened at least one $\mathfrak{c}_i + \mathfrak{d}_{bi+j}$ where $\mathfrak{d}_{bi+j}$ was well formed. The simulator can therefore extract a way to open $\mathfrak{c}_i$ since $\mathfrak{c}_i = (\mathfrak{c}_i + \mathfrak{d}_{bi+j}) - \mathfrak{d}_{bi+j}$. It will be able to compute polynomials f_l for $\mathfrak{c}_i$ with $\|f_l\|_\infty \leq 8 \cdot \delta \cdot \rho_l \cdot T \cdot \phi(m)$. Therefore, if some $\mathfrak{c}_i$ is not of this form, the prover can survive one iteration of the test with probability at most $t^{1-b} \cdot (e \cdot \ln(2))^{-b}$. To survive the entire protocol, the prover needs to win in at least one of the M iterations, and this happens with probability at most $M \cdot t^{1-b} \cdot (e \cdot \ln(2))^{-b}$, by the union bound.

Assume P_i is honest: Then when he decides whether to open a given ciphertext, the probability that a single coefficient is in range is $\frac{1}{4 \cdot \delta \cdot \phi(m) \cdot T}$. There are $4 \cdot \phi(m)$ coefficients in a single ciphertext and up to T ciphertexts to open, so by a union bound, P_i will not need to send $\perp$ at all, except with probability $1/\delta$. The probability that an honest prover fails to complete the protocol is hence $(1/\delta)^M$. We therefore see that the completeness error vanishes exponentially with increasing M, and in the soundness probability, we only loose $\log M$ bits of security.

It is easy to see that for each opening done by an honest prover, the polynomials z_l will have coefficients that are uniformly distributed in the expected range, so the protocol can be simulated.

Finally, note that in a normal run of the protocol, only 1 iteration is required, except with probability $1/\delta$. So in practice, what counts for the efficiency is the time we spend on one iteration.

In our experiments we implemented the above protocol with the following parameter choices $\delta = 256$, $M = 5$, $t = 12$ and $b = 16$. This guaranteed a cheating probability of 2^{-40}, as well as the probability of an honest prover failing of 2^{-40}. In addition the choice of $t = 12$ was to ensure that each run of the protocol created enough ciphertexts to be run in two executions of the main loop of the multiplication triple production protocol. By increasing t and decreasing b one can improve the amortized complexity of the protocol while keeping the error probabilities the same. This comes at the cost of increased memory usage, primarily because decreasing b to, e.g, $b/2$ means that t needs to be replaced by essentially t^2. On our test machines $t = 12$ seemed to provide the best compromise.

G Parameters of the BGV Scheme

In this appendix we present an analysis of the parameters needed by the BGV to ensure that the distributed decryption procedure can decrypt the ciphertexts produced in the offline phase and that the scheme is "secure". Unlike in [14], which presents the analysis in terms of a worst case analysis, we use the expected case analysis used in [16].

G.1 Expected Values of Norms

Given an element $a \in R$ (represented as a polynomial) we define $\|a\|_p$ to be the standard p-norm of the coefficient vector (usually for $p = 1, 2$ or ∞). We also define $\|a\|_p^{\mathsf{can}}$ to be the p-norm of the same element when mapped into the canonical embedding i.e.

$$\|a\|_p^{\mathsf{can}} = \|\kappa(a)\|_p$$

where $\kappa(a) : R \longrightarrow \mathbb{C}^{\phi(m)}$ is the canonical embedding. The key two relationships are that

$$\|a\|_\infty \leq c_m \cdot \|a\|_\infty^{\mathsf{can}} \text{ and } \|a\|_\infty^{\mathsf{can}} \leq \|a\|_1,$$

for some constant c_m depending on m. Since in our protocol we select m to be a power of two then we have $c_m = 1$.

We also define the *canonical embedding norm reduced modulo* q of an element $a \in R$ as the smallest canonical embedding norm of any a' which is congruent to a modulo q. We denote it as

$$|a|_q^{can} = \min\{\ \|a'\|_\infty^{\mathsf{can}} :\ a' \in R,\ a' \equiv a \pmod{q}\ \}.$$

We sometimes also denote the polynomial where the minimum is obtained by $[a]_q^{\mathsf{can}}$, and call it the *canonical reduction* of a modulo q.

Following [16][Appendix A.5] we examine the variances of the different distributions utilized in our protocol. Let ζ_m denote any complex primitive m-th root of unity. Sampling $a \in R$ from $\mathcal{HWT}(h, \phi(m))$ and looking at $a(\zeta_m)$ produces a random variable with variance h, when sampled from $\mathcal{ZO}(0.5, \phi(m))$ we obtain variance $\phi(m)/2$, when sampled from $\mathcal{DG}(\sigma^2, \phi(m))$ we obtain variance $\sigma^2 \cdot \phi(m)$ and when sampled from $\mathcal{U}(q, \phi(m))$ we obtain variance $q^2 \cdot \phi(m)/12$. By the law of large numbers we can use $6 \cdot \sqrt{V}$, where V is the above variance, as a high probability bound on the size of $a(\zeta_m)$, and this provides a bound on the canonical embedding norm of a.

If we take a product of two, three, or four such elements with variances $V_1, V_2, \ldots, V_4$ we use $16 \cdot \sqrt{V_1 \cdot V_2}$, $9.6 \cdot \sqrt{V_1 \cdot V_2 \cdot V_3}$ and $7.3 \cdot \sqrt{V_1 \cdot V_2 \cdot V_3 \cdot V_4}$ as the resulting bounds since

$$\mathsf{erfc}(4)^2 \approx \mathsf{erfc}(3.1)^3 \approx \mathsf{erfc}(2.7)^4 \approx 2^{-50}.$$

G.2 Key Generation

We first need to establish the rough distributions (i.e. variances) of the resulting keys arising from our key generation procedure. For our purposes we are only interested in the variance of the associated distributions in the canonical embedding, in which case we obtain

$$\begin{aligned}
\mathsf{Var}(\kappa(\mathfrak{s}_j)) &= n \cdot \mathsf{Var}(\kappa(\mathfrak{s}_{i,j})) = n \cdot h, \\
\mathsf{Var}(\kappa(a_j)) &= q_1^2 \cdot \phi(m)/12, \\
\mathsf{Var}(\kappa(\epsilon_j)) &= n \cdot \mathsf{Var}(\kappa(\epsilon_{i,j})) = n \cdot \sigma^2 \cdot \phi(m).
\end{aligned}$$

We will also need to analyze the distributions of the randomness needed to produce $\mathfrak{enc}_j$. Here we assume that all parties follow the protocol and we are only interested in the output final extended public key, thus we write (dropping the j to avoid overloading the reader)

$$\mathfrak{enc} = (b_{\mathfrak{s},\mathfrak{s}^2}, a_{\mathfrak{s},\mathfrak{s}^2})$$

where

$$b_{\mathfrak{s},\mathfrak{s}^2} = a_{\mathfrak{s},\mathfrak{s}^2} \cdot \mathfrak{s} + p \cdot e_{\mathfrak{s},\mathfrak{s}^2} - p_1 \cdot \mathfrak{s}^2.$$

We can also write

$$\begin{aligned}
\mathfrak{enc}_i' &= (b \cdot v_i + p \cdot e_{0,i} - p_1 \cdot \mathfrak{s}_i, a \cdot v_i + p \cdot e_{1,i}) \\
\mathfrak{zero}_i &= (b \cdot v_i' + p \cdot e_{0,i}', a \cdot v_i' + p \cdot e_{1,i}')
\end{aligned}$$

where $(v_i, e_{0,i}, e_{1,i}) \leftarrow \mathcal{RC}_s(0.5, \sigma^2, \phi(m))$ and $(v_i', e_{0,i}', e_{1,i}') \leftarrow \mathcal{RC}_s(0.5, \sigma^2, \phi(m))$. We therefore have

$$a_{\mathfrak{s},\mathfrak{s}^2} = \sum_{i=1}^{n} \mathfrak{s}_i \cdot \left(\sum_{j=1}^{n} a \cdot v_j + p \cdot e_{1,j} \right) + \sum_{i=1}^{n} (a \cdot v_i' + p \cdot e_{1,i}'),$$

and

$$\begin{aligned}
b_{\mathfrak{s},\mathfrak{s}^2} &= \sum_{i=1}^{n} \mathfrak{s}_i \cdot \left(\sum_{j=1}^{n} b \cdot v_j + p \cdot e_{0,j} - p_1 \cdot \mathfrak{s}_j \right) + \sum_{i=1}^{n} (b \cdot v'_i + p \cdot e'_{0,i}) \\
&= a_{\mathfrak{s},\mathfrak{s}^2} \cdot \mathfrak{s} - \mathfrak{s} \cdot \sum_{i=1}^{n} \mathfrak{s}_i \cdot \left(\sum_{j=1}^{n} a \cdot v_j + p \cdot e_{1,j} \right) + \sum_{i=1}^{n} \mathfrak{s}_i \cdot \left(\sum_{j=1}^{n} b \cdot v_j + p \cdot e_{0,j} - p_1 \cdot \mathfrak{s}_j \right) \\
&\quad + \sum_{i=1}^{n} \left((a \cdot \mathfrak{s} + p \cdot \epsilon) \cdot v'_i + p \cdot e'_{0,i} \right) - \mathfrak{s} \cdot \sum_{i=1}^{n} (a \cdot v'_i + p \cdot e'_{1,i}) \\
&= a_{\mathfrak{s},\mathfrak{s}^2} \cdot \mathfrak{s} + \sum_{i=1}^{n} \left(\sum_{j=1}^{n} b \cdot v_j \cdot \mathfrak{s}_i + p \cdot e_{0,j} \cdot \mathfrak{s}_i - p_1 \cdot \mathfrak{s}_i \cdot \mathfrak{s}_j - \mathfrak{s} \cdot \mathfrak{s}_i \cdot a \cdot v_j - \mathfrak{s} \cdot \mathfrak{s}_i \cdot p \cdot e_{1,j} \right) \\
&\quad + p \cdot \sum_{i=1}^{n} \left(\epsilon \cdot v'_i + e'_{0,i} - e'_{1,i} \cdot \mathfrak{s} \right) \\
&= a_{\mathfrak{s},\mathfrak{s}^2} \cdot \mathfrak{s} + p \cdot \sum_{i=1}^{n} \left(\sum_{j=1}^{n} (\epsilon \cdot v_j \cdot \mathfrak{s}_i + e_{0,j} \cdot \mathfrak{s}_i - \mathfrak{s} \cdot \mathfrak{s}_i \cdot e_{1,j}) + \epsilon \cdot v'_i + e'_{0,i} - e'_{1,i} \cdot \mathfrak{s} \right) - p_1 \cdot \mathfrak{s}^2 \\
&= a_{\mathfrak{s},\mathfrak{s}^2} \cdot \mathfrak{s} + p \cdot e_{\mathfrak{s},\mathfrak{s}^2} - p_1 \cdot \mathfrak{s}^2
\end{aligned}$$

where

$$e_{\mathfrak{s},\mathfrak{s}^2} = \sum_{i=1}^{n} \left(\sum_{j=1}^{n} (\epsilon \cdot v_j \cdot \mathfrak{s}_i + e_{0,j} \cdot \mathfrak{s}_i - \mathfrak{s} \cdot \mathfrak{s}_i \cdot e_{1,j}) + \epsilon \cdot v'_i + e'_{0,i} - e'_{1,i} \cdot \mathfrak{s} \right). \tag{2}$$

Thus the values $\mathfrak{enc}$ are indeed genuine "quasi-encryptions" of $-p_1 \cdot \mathfrak{s}^2$ with respect to the secret key $\mathfrak{s}$ and the modulus q_1. Equation 2 will be used later to establish the properties of the output of the SwitchKey procedure.

G.3 BGV Procedures

We can now turn to each of the procedures in turn of the two level BGV scheme we are using and estimate the output noise term. For a ciphertext $\mathfrak{c} = (c_0, c_1, \ell)$ we define the "noise" to be an upper bound on the value

$$\|c_0 - \mathfrak{s} \cdot c_1\|_\infty^{\mathsf{can}}.$$

$\underline{\mathsf{Enc}_{\mathfrak{pk}}(\mathbf{m})}$: Given a fresh ciphertext $(c_0, c_1, 1)$, we calculate a bound (with high probability) on the output noise by

$$\begin{aligned}
\|c_0 - \mathfrak{s} \cdot c_1\|_\infty &\leq \|c_0 - \mathfrak{s} \cdot c_1\|_\infty^{\mathsf{can}} \\
&= \|((a \cdot \mathfrak{s} + p \cdot \epsilon) \cdot v + p \cdot e_0 + \mathbf{m} - (a \cdot v + p \cdot e_1) \cdot \mathfrak{s}\|_\infty^{\mathsf{can}} \\
&= \|\mathbf{m} + p \cdot (\epsilon \cdot v + e_0 - e_1 \cdot \mathfrak{s})\|_\infty^{\mathsf{can}} \\
&\leq \|\mathbf{m}\|_\infty^{\mathsf{can}} + p \cdot \left(\|\epsilon \cdot v\|_\infty^{\mathsf{can}} + \|e_0\|_\infty^{\mathsf{can}} + \|e_1 \cdot \mathfrak{s}\|_\infty^{\mathsf{can}} \right) \\
&\leq \phi(m) \cdot p/2 + p \cdot \sigma \cdot \left(16 \cdot \phi(m) \cdot \sqrt{n/2} + 6 \cdot \sqrt{\phi(m)} + 16 \cdot \sqrt{n \cdot h \cdot \phi(m)} \right) = B_{\mathsf{clean}}.
\end{aligned}$$

Note this value of B_{clean} is different from that in [16] due to the different distributions resulting from the distributed key generation.

$\underline{\mathsf{SwitchModulus}((c_0, c_1), \ell)}$: If the input ciphertext has noise ν then the output ciphertext will have noise ν' where

$$\nu' = \frac{\nu}{p_\ell} + B_{\mathsf{scale}}.$$

The value B_{scale} is an upper bound on the quantity $\|\tau_0 + \tau_1 \cdot \mathfrak{s}\|_{\infty}^{\mathsf{can}}$, where $\kappa(\tau_i)$ is drawn from a distribution which is close to a complex Gaussian with variance $\phi(m) \cdot p^2/12$. We therefore, we can (with high probability) take the upper bound to be

$$\begin{aligned} B_{\mathsf{scale}} &= 6 \cdot p \cdot \sqrt{\phi(m)/12} + 16 \cdot p \cdot \sqrt{n \cdot \phi(m) \cdot h/12}, \\ &= p \cdot \sqrt{3 \cdot \phi(m)} \cdot \left(1 + 8 \cdot \sqrt{n \cdot h}/3\right). \end{aligned}$$

Again, note the dependence on n (compared to [16]) as the secret key $\mathfrak{s}$ is selected from a distribution with variance $n \cdot h$, and not just h. Also note the dependence on p due to the plaintext space being defined mod p as opposed to mod 2 in [16].

$\underline{\mathsf{Dec}_{\mathfrak{s}}(\mathfrak{c})}$: As explained in [14, 16] this procedure works when the noise ν associated with a ciphertext satisfies $\nu = c_m \cdot \nu < q_\ell/2$.

$\underline{\mathsf{DistDec}_{\mathfrak{s}_i}(\mathfrak{c})}$: The value B is an upper bound on the noise ν associated with a ciphertext we will decrypt in our protocols. To ensure valid distributed decryption we require

$$2 \cdot (1 + 2^{\mathsf{sec}}) \cdot B < q_\ell.$$

Given a value of B, we therefore will obtain a lower bound on p_0 by the above inequality. The addition of a random term with infinity norm bounded by $2^{\mathsf{sec}} \cdot B/(n \cdot p)$ in the distributed decryption procedure ensures that the individual *coefficients* of the sum $\mathbf{t}_1 + \cdots + \mathbf{t}_n$ are statistically indistinguishable from random, with probability $2^{-\mathsf{sec}}$. This does not imply that the adversary has this probability of distinguishing the simulated execution in [14] from the real execution; since each run consists of the exchange of $\phi(m)$ coefficients, and the protocol is executed many times over the execution of the whole protocol. We however feel that setting concentrating solely on the statistical indistinguishability of the coefficients is valid in a practical context.

$\underline{\mathsf{SwitchKey}(d_0, d_1, d_2)}$: In order to estimate the size of the output noise term we need first to estimate the size of the term

$$\|p \cdot d_2 \cdot \epsilon_{\mathfrak{s},\mathfrak{s}^2}\|_{\infty}^{\mathsf{can}}.$$

Using Equation 2 we find

$$\begin{aligned} \|p \cdot d_2 \cdot e_{\mathfrak{s},\mathfrak{s}^2})\|_{\infty}^{\mathsf{can}}/q_0 &\leq p \cdot \sqrt{\frac{\phi(m)}{12}} \cdot \Big[n^2 \cdot \sigma \cdot \Big(7.3 \cdot \sqrt{n \cdot h \cdot \phi(m)^2/2} + 9.6 \cdot \sqrt{h \cdot \phi(m)} \\ &\qquad + 7.3 \cdot h \cdot \sqrt{n \cdot \phi(m)}\Big) \\ &\qquad + n \cdot \Big(9.6 \cdot \sigma \cdot \sqrt{n \cdot \phi(m)^2/2} + 16 \cdot \sigma \cdot \sqrt{\phi(m)} \\ &\qquad + 7.6 \cdot \sigma \cdot \sqrt{\phi(m) \cdot n \cdot h}\Big)\Big] \\ &\leq p \cdot \phi(m) \cdot \sigma \cdot \Big[n^{2.5} \cdot (1.49 \cdot \sqrt{h \cdot \phi(m)} + 2.11 \cdot h) + 2.77 \cdot n^2 \cdot \sqrt{h} \\ &\qquad + n^{1.5} \cdot (1.96 \cdot \sqrt{\phi(m)} + 2.77 \cdot \sqrt{h}) + 4.62 \cdot n\Big] \\ &= B_{\mathsf{KS}}. \end{aligned}$$

Then if the input to SwitchKey has noise bounded by ν then the output noise value will be bounded by

$$\nu + \frac{B_{\mathsf{KS}} \cdot q_0}{p_1} + B_{\mathsf{scale}}.$$

$\underline{\mathsf{Mult}(\mathfrak{c}, \mathfrak{c}')}$: Combining the all the above, if we take two ciphertexts of level one with input noise bounded by ν and ν', the output noise level from multiplication will be bounded by

$$\nu'' = \left(\frac{\nu}{p_1} + B_{\mathsf{scale}}\right) \cdot \left(\frac{\nu'}{p_1} + B_{\mathsf{scale}}\right) + \frac{B_{\mathsf{KS}}}{p_1} + B_{\mathsf{scale}}.$$

G.4 Application to the Offline Phase

In all of our protocols we will only be evaluating the following circuit: We first add n ciphertexts together and perform a multiplication, giving a ciphertext with respect to modulus p_0 with noise

$$U_1 = \left(\frac{n \cdot B_{\mathsf{clean}}}{p_1} + B_{\mathsf{scale}} \right)^2 + \frac{B_{\mathsf{KS}} \cdot p_0}{p_2} + B_{\mathsf{scale}}.$$

We then add on another n ciphertexts, which are added at level one and then reduced to level zero. We therefore obtain a final upper bound on the noise for our adversarially generated ciphertexts of

$$U_2 = U_1 + \frac{n \cdot B_{\mathsf{clean}}}{p_1} + B_{\mathsf{scale}}.$$

To ensure valid (distributed) decryption, we require

$$2 \cdot U_2 \cdot (1 + 2^{\mathsf{sec}}) < p_0,$$

i.e. we take $B = U_2$ in our distributed decryption protocol.

This ensure valid decryption in our offline phase, however we still need to select the parameters to ensure security. Following the analysis in [16] of the BGV scheme we set, for 128-bit security,

$$\phi(m) \geq 33.1 \cdot \log \left(\frac{q_1}{\sigma} \right).$$

Combining the various inequalities together; a search of the parameter space the fixed values of $\sigma = 3.2$, $\mathsf{sec} = 40$ and $h = 64$, for several choices of p, n yields the estimates in tables 4, 5 and 6. And it is these parameter sizes which we use to generate the primes and rings in our implementation.

n	$\phi(m)$	$\log_2 p_0$	$\log_2 p_1$	$\log_2 q_1$	$\log_2(U_2)$
2	8192	130	104	234	89
3	8192	132	104	236	90
4	8192	132	104	236	91
5	8192	132	106	238	90
6	8192	132	106	238	91
7	8192	132	108	240	91
8	8192	132	108	240	91
9	8192	132	110	242	91
10	8192	132	110	242	91
20	8192	134	110	244	93
50	8192	136	114	250	94
100	8192	136	116	252	95

Table 4. Parameters for $p \approx 2^{32}$.

n	$\phi(m)$	$\log_2 p_0$	$\log_2 p_1$	$\log_2 q_1$	$\log_2(U_2)$
2	16384	196	136	332	154
3	16384	196	138	334	154
4	16384	196	140	336	155
5	16384	196	142	338	155
6	16384	198	140	338	156
7	16384	198	140	338	156
8	16384	198	140	338	157
9	16384	198	142	340	156
10	16384	198	142	340	156
20	16384	198	146	344	157
50	16384	200	148	348	158
100	16384	202	150	352	160

Table 5. Parameters for $p \approx 2^{64}$.

n	$\phi(m)$	$\log_2 p_0$	$\log_2 p_1$	$\log_2 q_1$	$\log_2(U_2)$
2	32768	324	202	526	283
3	32768	326	202	528	285
4	32768	326	204	530	284
5	32768	326	204	530	285
6	32768	326	206	532	284
7	32768	326	206	532	285
8	32768	326	208	534	285
9	32768	326	208	534	285
10	32768	326	208	534	285
20	32768	328	210	538	286
50	32768	330	212	542	289
100	32768	330	216	546	288

Table 6. Parameters for $p \approx 2^{128}$.