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. 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. 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.

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 [18] and that of many-party protocols, based on secret sharing techniques [5,9,22]. Only in recent years work has shifted to achieve active security [16,17,21], 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

J. Crampton, S. Jajodia, and K. Mayes (Eds.): ESORICS 2013, LNCS 8134, pp. 1–18, 2013.

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 [13] (dubbed SPDZ, and pronounced Speedz). In [13] 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 [11] 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 [21,10]).

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 [11,13] 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 [15]. This new sub-protocol may be of independent interest in other applications which require distributed decryption in an SHE/FHE scheme.
3. In [11] 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 [13] 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 [20]. This method has running time similar to the ZKPoK approach utilized in [13], 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 [13], 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 [13]: it has full active security assuming access to a functionality for key generation. However, it has enhanced functionality and performance, compared to [13], 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 [11] 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.

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 [11,13] 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. 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 [12].

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 [13], 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 [1,7,8].

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 [6,14,15] 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 which can be thought of as a function of the underlying security parameter. 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 [13,15] 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 [15] 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 [13]; 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.

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 [15]; note we need the more complex function of Appendix E of the full version of [15] 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 [15]. 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 [13], since our final ciphertexts consist of only two elements as opposed to three in [13]. 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 the full version of this abstract [12]. 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 [6,15]. 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 $\mathfrak{c}' = ((c_0', c_1'), 1)$. 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

The protocol for distributed key generation protocol is given in Figure 1. It makes use of an abstract functionality $\mathcal{F}_{\text{COMMIT}}$ which implements a commitment functionality. In practice this functionality is implemented in the random oracle model via hash functions, see the full version for details [12]. Here we present a high level overview.

As remarked in the introduction, the authors of [13] 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:

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. 1. The protocol for key generation.

we augment the public key with the keyswitching data from [15] 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 [15], 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 1 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 the full version [12]. 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 [15] 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 1 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.*

$\mathcal{F}_{\text{KEYGEN}}$ 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. See the full version for a complete technical discription of $\mathcal{F}_{\text{KEYGEN}}$.

The BGV cryptosystem resulting from $\mathcal{F}_{\text{KEYGEN}}$ is proven semantically secure by the following theorem from the full version of this paper [12].

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 mentioned earlier.*

4.2 EncCommit

We use a sub-protocol $\Pi_{\text{ENCCOMMIT}}$ to replace the Π_{ZKPoPK} protocol from [13]. 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 the full version of this paper 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 $\Pi_{\text{ENCCOMMIT}}$ is presented in Figure 2. A proof of the following theorem, and a description of the associated ideal functionality, are given in the full version of this paper [12].

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 the full version, we would get a version of $\mathcal{F}_{\text{SHE}}$ where the adversary is not allowed to attempt cheating.

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

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 [13]. 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 [13], 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 [13], 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 3; it is not intended to implement any functionality – it is just a procedure that can be called in both the offline and online phases.

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.

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. 3. Method to Check MACs on Partially Opened Values

MACCheck has the following important properties.

Lemma 1. *The protocol* 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 the full version of this paper.

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 $\mathbf{c}_\alpha$ 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 a variant of the method from [13] (precise details are in the full version of this paper). The two key differences between our protocol and that of [13], is that

1. The expensive ZKPoKs, used to verify that ciphertexts encrypting random values are correctly produced, are replaced with our protocol $\Pi_{\text{EncCommit}}$.
2. We generate squaring tuples and shared bits, as well as multiplication triples.

The tuple production protocol can be run repeatedly, alongside the tuple-checking subphase 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 [13], 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 [13]); 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 the full version of this paper [12].

We show that the resulting protocol Π_{PREP}, 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. Full details of $\mathcal{F}_{\text{PREP}}$ and Π_{PREP} are given in the full version, along with 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 [13]; 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 [13], since the linear combination of partially opened values is truly random in our case, while it has lower entropy in [13].

In the full version of the paper we present the protocol Π_{ONLINE}, which is the obvious adaption of the equivalent protocol from [13]. In addition we present an ideal functionality $\mathcal{F}_{\text{ONLINE}}$ and prove the following theorem.

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.*

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 the full version 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 [19] 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.

We compare the results to that obtained in [11], since no other protocol can provide malicious/covert security for $t < n$ corrupted parties. In the case of covert security the authors of [11] 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 [11] 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 [11], which used the method

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

n	$p \approx$	c	Run Times KeyGen	Run Times Offline	Time per 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

n	$p \approx$	c	Run Times KeyGen	Run Times Offline	Time per 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

from [13]. 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.

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

	$n = 2$		$n = 3$	
$p \approx$	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

By way of comparison for a prime of 64 bits the authors of [11] 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 [11] 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 [13] 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 this by a factor of about 10, then we would get results better than those of [13,11], all other things being equal. More information can be found in the full version [12].

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.

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 [13] gave an amortized time of 20000 multiplications per second over a 64-bit prime field, with three players.

Table 3. Online Times

		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

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[1] , and by a Royal Society Wolfson Merit Award.

References

1. Aliasgari, M., Blanton, M., Zhang, Y., Steele, A.: Secure computation on floating point numbers. In: Network and Distributed System Security Symposium, NDSS 2013. Internet Society (2013)
2. Aumann, Y., Lindell, Y.: Security against covert adversaries: Efficient protocols for realistic adversaries. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 137–156. Springer, Heidelberg (2007)
3. Aumann, Y., Lindell, Y.: Security against covert adversaries: Efficient protocols for realistic adversaries. J. Cryptology 23(2), 281–343 (2010)
4. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 420–432. Springer, Heidelberg (1992)

[1] 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.

5. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 192–206. Springer, Heidelberg (2008)
6. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: ITCS, pp. 309–325. ACM (2012)
7. Catrina, O., Saxena, A.: Secure computation with fixed-point numbers. In: Sion, R. (ed.) FC 2010. LNCS, vol. 6052, pp. 35–50. Springer, Heidelberg (2010)
8. Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In: Halevi, S., Rabin, T. (eds.) TCC 2006. LNCS, vol. 3876, pp. 285–304. Springer, Heidelberg (2006)
9. Damgård, I., Geisler, M., Krøigaard, M., Nielsen, J.B.: Asynchronous multiparty computation: Theory and implementation. In: Jarecki, S., Tsudik, G. (eds.) PKC 2009. LNCS, vol. 5443, pp. 160–179. Springer, Heidelberg (2009)
10. Damgård, I., Keller, M.: Secure multiparty AES. In: Sion, R. (ed.) FC 2010. LNCS, vol. 6052, pp. 367–374. Springer, Heidelberg (2010)
11. Damgård, I., Keller, M., Larraia, E., Miles, C., Smart, N.P.: Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. In: Visconti, I., De Prisco, R. (eds.) SCN 2012. LNCS, vol. 7485, pp. 241–263. Springer, Heidelberg (2012)
12. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority – or: Breaking the SPDZ limits (2012)
13. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer, Heidelberg (2012)
14. Gentry, C., Halevi, S., Smart, N.P.: Fully homomorphic encryption with polylog overhead. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 465–482. Springer, Heidelberg (2012)
15. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 850–867. Springer, Heidelberg (2012)
16. Kreuter, B., Shelat, A., Shen, C.-H.: Towards billion-gate secure computation with malicious adversaries. In: USENIX Security Symposium 2012, pp. 285–300 (2012)
17. Lindell, Y., Pinkas, B., Smart, N.P.: Implementing two-party computation efficiently with security against malicious adversaries. In: Ostrovsky, R., De Prisco, R., Visconti, I. (eds.) SCN 2008. LNCS, vol. 5229, pp. 2–20. Springer, Heidelberg (2008)
18. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay - Secure two-party computation system. In: USENIX Security Symposium 2004, pp. 287–302 (2004)
19. Montgomery, P.L.: Modular multiplication without trial division. Math. Comp. 44, 519–521 (1985)
20. Nielsen, J.B., Nordholt, P.S., Orlandi, C., Burra, S.S.: A new approach to practical active-secure two-party computation. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 681–700. Springer, Heidelberg (2012)
21. Pinkas, B., Schneider, T., Smart, N.P., Williams, S.C.: Secure two-party computation is practical. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 250–267. Springer, Heidelberg (2009)
22. SIMAP Project. SIMAP: Secure information management and processing, `http://alexandra.dk/uk/Projects/Pages/SIMAP.aspx`