Actively Secure Setup for SPDZ

Abstract. We present an actively secure, practical protocol to generate the distributed secret keys needed in the SPDZ offline protocol. The resulting distribution of the public and secret keys is such that the associated SHE 'noise' analysis is the same as if the distributed keys were generated by a trusted setup. We implemented the presented protocol for distributed BGV key generation within the `SCALE-MAMBA` framework. Our method makes use of a new method for creating doubly (or even more) authenticated bits in different MPC engines, which has applications in other areas of MPC-based secure computation. We were able to generate keys for two parties and a plaintext size of 64 bits in around five minutes, and approximately twenty minutes for a 128 bit prime.

Keywords: MPC · Somewhat Homomorphic Encryption · Key Generation

Note: A prior version of this paper was submitted to Euro-Crypt. The current version is much improved in terms of the algorithms used and the performance. Thus if you reviewed this before please look again. We do not include the original reviews as the algorithms are now very different.
This note is included a per the IACR policy on 'sticky reviews'

1 Introduction

The SPDZ protocol for Multi-Party Computation (MPC) was introduced in 2012 [11]. This protocol is in the pre-processing family of protocols which are actively secure-with-abort for a dishonest majority of participants. Due to many improvements over the intervening years it provides a highly efficient mechanism to perform MPC for an arbitrary number of participants. However, the protocol comes with a major security issue: namely that it seems to require a trusted setup. This trusted setup is the creation of a public key for the Brakerski-Gentry-Vaikuntanathan [7] (BGV) homomorphic encryption scheme in which the private key is securely distributed amongst the n-parties.

In the original SPDZ paper [11] this was assumed to come from some trusted setup. In the follow up paper [10] a covertly secure protocol for generating a suitably distributed set of private keys, and the associated public key was introduced. However, this came with a number of disadvantages, as well as the reduction to just covert security. In particular the distributions of the underlying public keys were different from those one could attain via a trusted setup, which led to a more complicated noise analysis, and larger parameters.

In subsequent works the issue of the setup of the public key for the BGV encryption scheme has been dismissed as a setup assumption, which could either be performed in a live system using trusted hardware or via another MPC protocol. Given the complexity of the covertly secure key generation protocol from [10] it has always been assumed that the key generation for SPDZ would require a *complex* MPC protocol to perform it. In [4] the authors present a key generation method for a distributed SHE scheme using various Σ-protocols. To our knowledge this has never been implemented, and the methodology again produces a key generation which is different from what would be done via a trusted setup. In [21] a passively secure distributed key generation method is used for threshold SHE schemes, again producing a distribution different from that one would have in a purely trusted setup.

In this paper we show that we can obtain actively secure distributed key generation, with virtually identical secret key distributions as in the trusted setup case. In particular the noise analysis for the resulting public key is identical to that one would have if using a trusted setup. Our protocol is also relatively simple, although it does make use of complex generic MPC technology. In particular, our protocol generates a public/private key with exactly the same distribution as the ideal trusted setup does in the `SCALE-MAMBA` system[1], bar the fact we generate secret keys with *expected* Hamming weight h as opposed to *exact* Hamming weight h.

We are also able to generate secret keys from binomial distributions, which can be seen as approximate Gaussian error distributions. These lead to slightly larger parameters, but do not suffer from the security concerns that low Hamming weight secret distributions have [9]. In addition, for our purposes, using such keys produces a faster distributed key generation procedure. The effect of using such keys makes the ring parameters slightly bigger, but it decreases the runtime of distributed key generation by about a half.

Our protocol makes use of a generic MPC functionality for actively secure MPC-with-abort for dishonest majorities over a finite field. This might seem to imply that we require SPDZ to create SPDZ, however this circular dependency is removed by utilizing either the BDOZ protocol [5] or the SPDZ protocol executed with the MASCOT pre-processing phase [19]. The first of these, BDOZ, makes use of n public keys for a linear homomorphic encryption scheme where one private key is held by each player. The second option, MASCOT, is based on Oblivious Transfer. Both of these *base* MPC protocols are not as efficient as the SPDZ protocol based on homomorphic encryption, but we will only be using the base protocols for the one-time setup phase for SPDZ. In particular the underlying generic MPC protocol that we will use for key generation is $O(n^2)$ in complexity; but we use this to create the distributed secret keys for an MPC protocol which has complexity $O(n)$. To avoid confusion we will refer to SPDZ with a MASCOT based pre-processing as MASCOT-SPDZ, where as when we talk about SPDZ we mean the pure SPDZ protocol with a pre-processing based on homomorphic encryption.

The overall construction of our protocol is based on four key observations; all of which are relatively simple. Firstly, the generation of the public key data

[1] We use `SCALE-MAMBA` as a reference work throughout this paper as it gives a fixed target (including key sizes) for the final distributed keys we are trying to produce.

given the secret key data and randomness for a BGV public key is essentially a linear operation and thus comes for free in LSSS based MPC protocols such as BDOZ and MASCOT-SPDZ. Secondly, the BGV public key for SPDZ is a two level BGV scheme thus the ciphertext modulus q needed to construct the BGV public key is a product of two primes $q = p_0 \cdot p_1$. In particular the public key is simply the lift to modulo q of the public key modulo p_0 and p_1, performed via the Chinese Remainder Theorem (CRT). If we select p_0 and p_1 to be prime, as `SCALE-MAMBA` does, then we can use two MPC systems (one over p_0 and one over p_1) to perform the operations, and then obtain the final result via application of the CRT. We assume these two MPC systems come as ideal functionalities $\mathcal{F}_{\mathsf{MPC}}^{p_0}$ and $\mathcal{F}_{\mathsf{MPC}}^{p_1}$. Thirdly, all the random values required in BGV key generation can be boiled down to the generation of random bits, which are then processed in various ways. Thus a key issue is how to generate these random bits. Whilst BDOZ and MASCOT-SPDZ can be adapted to produce authenticated bits as part of their pre-processing, using much the same trick as proposed in [10], this will produce *different* random bits in $\mathcal{F}_{\mathsf{MPC}}^{p_0}$ and $\mathcal{F}_{\mathsf{MPC}}^{p_1}$. Thus our fourth, and final, observation is that we can produce sharings of the same random bit in both $\mathcal{F}_{\mathsf{MPC}}^{p_0}$ and $\mathcal{F}_{\mathsf{MPC}}^{p_1}$ using an adaption of the daBit method from [3] and [24].

Indeed our new method for daBit is more general and more efficient than the method presented in [3,24]. We require a daBit method which works for two large primes, whereas [3,24] require a method for a large prime and a small prime (in particular two). Our new method deals with any prime size for the two MPC engines, can be extended to more MPC engines than just two, and is built upon an abstraction which allows it to be used with any form of LSSS based MPC engine in the SPDZ family (e.g. BDOZ, MASCOT or SPDZ itself).

The most expensive part of the daBit generation procedure for producing daBits with active security in [3,24] was in verifying consistency of the daBits between the two instances of $\mathcal{F}_{\mathsf{MPC}}$. The idea in these works was to check the same random linear combinations of bits in both instances simultaneously, which was challenging because one field had characteristic 2 and the other some large prime p which meant the XOR had to be emulated in the prime field, requiring multiplication in MPC. In addition, to generate a bit, one needed to perform XOR in both of the fields under consideration. Our observation in this work is that we use an auxillary MPC engine for a large prime p to generate bits using the standard square-root trick for generating bits, these are then mapped into the target MPC engines. The auxillary MPC engine is used to obtain a subset sum over the integers, which is then compared to the equivalent subset sum in the target MPC engines. Security now reduces to a variant of the Multiple Subset-Sum Problem[2].

We decided to use MASCOT-SPDZ as the underlying MPC protocol for the BGV key generation, with our implementation building upon the already exist-

[2] Carsten Baum has pointed out that we can remove this reduction to the subset-sum by increasing the number of bits we throw away. This however results in a less efficient protocol, thus we rely on the Multiple Subset-Sum Problem to obtain an efficient protocol.

ing code-base for OT present in the `SCALE-MAMBA` framework. In addition we ran experiments with different values for the standard deviation of the centred binomial distribution, and experiments between Hamming weight restricted secret keys and secret keys generated from a centred binomial distribution. In the fastest case, of standard deviation $\sigma = \sqrt{2} = 0.707$ for the centred binomial distribution and FHE keys distributed following this same distribution, for two parties and a 64 bit plaintext modulus our results show that we can distributively generate BGV keys in around five minutes. We ran experiments for 64 and 128 bit primes for the plaintext space for two and three parties for all settings; and for our fastest settings we also ran experiments for four and five parties.

For the parameters used in SCALE-MAMBA, which is $\sigma = \sqrt{10} = 3.16$ and Hamming weight limited secret keys, we find a key generation time of 47 minutes for two parties and a 64 bit modulus. We give a detailed report of our implementation in Section 6, in which the triple generation throughput and the shared bit throughput are given for all our test cases.

We end this introduction by noting that in [6] a method to perform the SPDZ offline phase using no-communication is presented. However, this method is impractical as currently presented. The method still requires a distributed decryption capability of the underlying SHE scheme. Thus to use this work even in theory one needs to be able to generate such distributed keys in a secure manner, such as this work enables. We also note that using the silent-OT method of [6] one may be able to achieve better runtimes. The paper reports that they can achieve 600,000 correlated OT's per second. However, due to the increased computational costs of the silent-OT method this might not translate to the LAN setting in our experiments.

2 Preliminaries

In this section we provide the necessary background on the type of BGV public key we need to produce, as well as the underlying distributions and the base MPC protocols we will be using.

2.1 Cyclotomic Rings and Distributions over such Rings

The BGV encryption scheme is defined over a cyclotomic ring $R = \mathbb{Z}[X]/(X^N + 1)$, where for our purposes we take N to be a power of two. Thus $X^N + 1$ is the $m = 2 \cdot N$-th cyclotomic polynomial, and $N = \phi(m)$. We let $\odot$ denote the multiplication operation in R.

Following [15][Full version in [14], Appendix A.5] the `SCALE-MAMBA` system utilizes the following distributions in the key generation procedure.

- $\mathsf{HWT}(h, N)$: 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.

- $\mathsf{dN}(\sigma^2, N)$: This generates a vector of length N with elements chosen according to an *approximation* to the discrete Gaussian distribution with variance σ^2, by sampling from a centered binomial distribution.
- $\mathsf{U}(q, N)$: This generates a vector of length N with elements generated uniformly modulo q.

In particular for the distribution $\mathsf{dN}(\sigma^2, N)$ `SCALE-MAMBA` approximates $\mathsf{dN}(\sigma^2, N)$ using the approximation from [1]. In particular $\mathsf{dN}(\sigma^2, N)$ is replaced by the centered binomial distribution where elements are returned using the formula

$$c_j = \sum_{i=0}^{k} b_{2\cdot i} - b_{2\cdot i+1}$$

for uniformly random bits $b_j \in \{0, 1\}$ for $j = 0, \ldots, 2 \cdot k - 1$. The default settings of `SCALE-MAMBA` use $k = 20$, giving us $\sigma = \sqrt{k/2} = \sqrt{10} = 3.16$.

We make a small change to one of the above distributions in our work. The distribution $\mathsf{HWT}(h, N)$ is used to sample the secret key, where in [15] (and in `SCALE-MAMBA`) the value h is selected to be a power of two; in particular $h = 64$. In our work we replace $\mathsf{HWT}(h, N)$ with the distribution which picks each coefficient with respect to the Bernoulli distribution $B(h/N)$. Thus we use the approximation $\mathsf{HWT}(h, N) \approx B(h/N)^N$. The Hamming weight of the vectors output by this distribution follows a binomial distribution with mean h. We still use $h = 64$ in our recommended construction though. The "noise analysis" behind the homomorphic operations used in the SPDZ protocol are easily checked not to be affected by this change, and in addition the security arguments for using low Hamming weight secret keys (as discussed in [15]) are also not affected. In particular the noise analysis used in [15] or `SCALE-MAMBA`is an 'average case' analysis in the key generation. Thus the standard deviation in the canonical norm of the secret key is $\sqrt{h}$ if an exact Hamming weight of h is used. It is this standard deviation which is the contributing term in the noise analysis. If one generates the secret key using only an expected Hamming weight then you obtain the same standard deviation; thus nothing changes in the analysis by using our slightly different secret key distribution.

2.2 The BGV Key Generation Procedure

For a modulus q we let R_q denote the above ring localised at the modulus q, i.e. $R_q = (\mathbb{Z}/q\mathbb{Z})[X]/(X^N + 1)$. The SPDZ protocol requires a two-leveled scheme with moduli p_0 and p_1 with $q_1 = p_0 \cdot p_1$ and $q_0 = p_0$. We require, for efficiency, that

$$\begin{aligned} p_1 &\equiv 1 \pmod{p}, \\ p_0 - 1 \equiv p_1 - 1 \equiv p - 1 &\equiv 0 \pmod{N}, \end{aligned}$$

where p is the plaintext modulus. The moduli p_0 and p_1 are selected to be distinct primes. In which case, by the CRT, we have $R_q \cong R_{p_0} \times R_{p_1}$. In addition, due to the above restrictions on the primes p_0 and p_1, there is an efficient FFT algorithm on R_{p_i}, which requires no extension field arithmetic. Thus one can efficiently multiply in R_{p_i} by executing

$$\mathbf{a} \odot \mathbf{b} = \mathsf{FFT}^{-1}(\mathsf{FFT}(\mathbf{a}) \cdot \mathsf{FFT}(\mathbf{b}))$$

where $\cdot$ here is the component wise product. Note that the FFT operation is a *linear* operation and thus can be executed in an MPC engine for free. These facts we shall use in our distributed key generation protocol.

The BGV public key is of the form $(\mathbf{a}, \mathbf{b}) \in R_q$ where

$$\mathbf{a} \leftarrow \mathsf{U}(q, N) \quad \text{and} \quad \mathbf{b} = \mathbf{a} \odot \mathfrak{sk} + p \cdot \mathbf{e}$$

where $\mathbf{e} \leftarrow \mathsf{dN}(\sigma^2, N)$. The secret key $\mathfrak{sk}$ for our purposes will be selected from the distribution $B(h/N)^N$. We also require, for the SPDZ protocol, the switching key data $(\mathbf{a}_{\mathfrak{sk},\mathfrak{sk}^2}, \mathbf{b}_{\mathfrak{sk},\mathfrak{sk}^2})$ which is of the form

$$\mathbf{a}_{\mathfrak{sk},\mathfrak{sk}^2} \leftarrow \mathsf{U}(q, N) \quad \text{and} \quad \mathbf{b}_{\mathfrak{sk},\mathfrak{sk}^2} = \mathbf{a}_{\mathfrak{sk},\mathfrak{sk}^2} \odot \mathfrak{sk} + p \cdot e_{\mathfrak{sk},\mathfrak{sk}^2} - p_1 \cdot \mathfrak{sk}^2$$

where $\mathbf{e}_{\mathfrak{sk},\mathfrak{sk}^2} \leftarrow \mathsf{dN}(\sigma^2, N)$.

The goal in a distributed key generation protocol for the SPDZ system is to output the public values $\mathfrak{pk} = (\mathbf{a}, \mathbf{b}, \mathbf{a}_{\mathfrak{sk},\mathfrak{sk}^2}, \mathbf{b}_{\mathfrak{sk},\mathfrak{sk}^2})$ to all players, whilst player P_i obtains a value $\mathfrak{sk}_i \in R_q$ such that

$$\mathfrak{sk} = \mathfrak{sk}_1 + \ldots + \mathfrak{sk}_n \pmod{q}.$$

We also require that no party can influence the choice of secret key, and no proper subset of the n parties can deduce any information about the secret key, bar what can be deduced from the public key. Thus we aim to create a protocol which securely realizes the functionality given in Figure 1, where $\mathsf{ParamGen}(1^\kappa, \log_2 p, n)$ is a function which produces the system parameters (p, p_0, p_1).

However, due to the concerns raised in [9] in relation to low Hamming weight keys, we also examine the case of secret keys generated by a centred binomial distribution; namely when we select $\mathfrak{sk}$ from $\mathsf{dN}(\sigma^2, N)$. These lead to slightly larger parameters for the underlying FHE systems, but the method to produce the keys is simpler.

2.3 Base MPC Protocols

In Figure 2 we present the MPC functionality for our base MPC protocols, either BDOZ or MASCOT-SPDZ in the case where we are generating keys or SPDZ when we are doing traditional daBit generation.

To simplify presentation of protocols using this functionality we shall represent a value held in the memory of such an MPC functionality by $\langle x \rangle_p$, and then addition and multiplication of such elements will be represented by

$$\langle x \rangle_p + \langle y \rangle_p, \qquad \langle x \rangle_p \cdot \langle y \rangle_p.$$

Functionality $\mathcal{F}_{\mathsf{KeyGen}}$

1. When receiving the message *start* from all honest parties, run $P \leftarrow \mathsf{ParamGen}(1^\kappa, \log_2 p, n)$, and then, using the parameters generated, run $(\mathfrak{pk}, \mathfrak{sk}) \leftarrow \mathsf{KeyGen}()$ (recall P, and hence 1^κ, is an implicit input to all functions we specify). Send $\mathfrak{pk} = (\mathbf{a}, \mathbf{b}, \mathbf{a}_{\mathfrak{sk},\mathfrak{sk}^2}, \mathbf{b}_{\mathfrak{sk},\mathfrak{sk}^2})$ to the adversary.
2. Receive from the adversary a set of shares $\mathfrak{sk}_j \in R_q$ for each corrupted party P_j for $j \neq 1$.
3. Construct a complete set of shares $(\mathfrak{sk}_1, \ldots, \mathfrak{sk}_n)$ consistent with the adversary's choices and $\mathfrak{sk}$. This is done be selecting $\mathfrak{sk}_i$ uniformly at random for honest i, subject to the constraint that $\mathfrak{sk} = \sum \mathfrak{sk}_i$. Note that this is always possible since the corrupted players form an unqualified set.
4. The functionality waits for an input from the environment.
5. If this input is $\mathsf{Deliver}$ then send $\mathfrak{pk}$ to all players and $\mathfrak{sk}_i$ to each honest player P_i, and send $\mathfrak{sk}_1$ to player P_1 if P_1 is dishonest.
6. If the adversarial input is not equal to $\mathsf{Deliver}$ then abort.

Figure 1. The Ideal Functionality for Key Generation (Adapted from [11])

For inputing and outputting values to/from a player/all players we will write

$$\langle x \rangle_p \leftarrow \mathsf{Input}(P_i), \qquad P_i \leftarrow \mathsf{Output}(\langle x \rangle_p), \qquad x \leftarrow \mathsf{Open}(\langle x \rangle_p).$$

That the BDOZ, MASCOT-SPDZ and SPDZ protocols implement such a functionality securely can be found proved in the respective papers [5], [19] and [11].

In such MPC protocols addition, and in fact any linear operation, is a 'free' operation, whereas multiplication will be assumed to take a single 'time' unit of operation. Another metric one often examines is the round complexity, in this case a multiplication takes one round, but multiplications which can be performed in parallel also only take one round of operation. Inputing, outputting or opening a data item also requires one round of communication, and such operations can be performed in parallel.

2.4 The $\mathcal{F}^B_{\mathsf{Rand}}(M)$ Functionality

We also require a functionality $\mathcal{F}^B_{\mathsf{Rand}}(M)$ which allows the parties to agree on M random values in the range $[0, \ldots, B)$. In practice this can be implemented by all parties committing to a seed, then the parties open the seeds. The seeds are then XOR'd together to produce a single shared seed, which is passed as the key to a PRF to produce the shared random values. We present this as an ideal functionality in Figure 3.

3 maBits: Generating Multiply Authenticated Bits

The main problem in performing actively secure key generation for SPDZ is to produce secure randomly shared bits within the MPC functionalities; in which

Functionality $\mathcal{F}_{\mathsf{MPC}}^{p}$

The functionality runs with parties $P_1, \ldots, P_n$ and an ideal adversary $\mathcal{A}$. Let A be the set of corrupt parties. Given a set I of valid identifiers, all values are stored in the form $(varid, x)$, where $varid \in I$.

Initialize: On input $(Init, p)$ from all parties, with p a prime, the functionality stores p. The adversary is assumed to have statically corrupted a subset $\mathcal{A}$ of the parties.

Input: This takes input $(Input, P_i, varid, x)$ from P_i, with $x \in \mathbb{F}_p$, and $(input, P_i, varid, ?)$ from all other parties, with $varid$ a fresh identifier. If the $varid$'s are the same the functionality stores $(varid, x)$, otherwise it aborts.

Add: On command $(Add, varid_1, varid_2, varid_3)$ from all parties:

1. If $varid_1, varid_2$ are not present in memory or $varid_3$ is then the functionality aborts.
2. 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:

1. If $varid_1, varid_2$ are not present in memory or $varid_3$ is then the functionality aborts.
2. The functionality retrieves $(varid_1, x)$, $(varid_2, y)$ and stores $(varid_3, x \cdot y)$.

Output: On input $(Output, varid, i)$ from all parties (if $varid$ is present in memory),

1. The functionality retrieves $(varid, y)$.
2. If $i = 0$ then the functionality outputs y to the environment, otherwise it outputs $\perp$ to the environment.
3. The functionality waits for an input from the environment.
4. If this input is $\mathsf{Deliver}$ then y is output to all players if $i = 0$, or y is output to player i if $i \neq 0$.
5. If the adversarial input is not equal to $\mathsf{Deliver}$ then abort.

Figure 2. The ideal functionality for MPC with Abort over $\mathbb{F}_p$

Functionality $\mathcal{F}_{\mathsf{Rand}}^{B}(M)$

1. On input $(\mathsf{Rand}, \mathrm{cnt})$ from all parties, if the counter value is the same for all parties and has not been used before, the functionality samples $r_i \leftarrow [0, \ldots, B)$ for $i = 1, \ldots, M$.
2. The values r_i are sent to the adversary, and the functionality waits for an input.
3. If the input is $\mathsf{Deliver}$ then the values r_i are sent to all parties, otherwise the functionality aborts.

Figure 3. The ideal $\mathcal{F}_{\mathsf{Rand}}^{B}(M)$ functionality

the bit is zero with probability $1/2$, and one with probability $1/2$. The 'standard' trick to do this, borrowed from [10], for a *single* MPC functionality is given in Figure 4[3]. However, if we execute this procedure with respect to both $\mathcal{F}_{\mathsf{MPC}}^{p_0}$ and $\mathcal{F}_{\mathsf{MPC}}^{p_1}$ then we will obtain two shared random bits $\langle b_0 \rangle_{p_0}$ and $\langle b_1 \rangle_{p_1}$ but we will not necessarily have $b_0 = b_1$.

Entering a Random Bit $\Pi_{\mathsf{MPC}}^{p}.\mathsf{GenBit}()$

```
1. For $i = 1, \ldots, n$ execute $\langle x_i \rangle_p \leftarrow \mathsf{Input}(P_i)$, where $x_i$ is a random element in $\mathbb{F}_p$.
2. $\langle x \rangle_p \leftarrow \sum_{i=1}^{n} \langle x_i \rangle_p$.
3. $\langle y \rangle_p \leftarrow \langle x \rangle_p \cdot \langle x \rangle_p$.
4. $y \leftarrow \mathsf{Open}(\langle y \rangle_p)$.
5. If $y = 0$ then restart the process.
6. $z \leftarrow \sqrt{y}$, picking the value $z \in [0, \ldots, p/2)$.
7. $\langle a \rangle_p \leftarrow \langle x \rangle_p / z$.
8. $\langle b \rangle_p \leftarrow (\langle a \rangle_p + 1)/2$.
9. Return $\langle b \rangle_p$.
```

Figure 4. 'Standard' method to produce a shared random bit in Π_{MPC}^{p}

To obtain shared random bits in the two MPC systems which are *identical* we need to adapt the daBit idea from [24]. In this paper it is shown how to obtain identical shared random bits in two MPC systems; one being a SPDZ-like system modulo p, and one being a BDOZ-like system over $\mathbb{F}_2$ based on OT for garbled circuit style computations. In our key generation protocol we require shared random bits in two SDPZ-like systems for large moduli p_0 and p_1. This makes the protocol to generate the shared random bits a little easier to understand than the one considered in [24]. Indeed we present a more general protocol than that which is needed for our key generation method. Our new method includes the case considered in [24], and is more efficient than the improved method considered in [3].

For our generalisation we consider a set of t SPDZ-like MPC systems with moduli $p_0, \ldots, p_{t-1}$. Our goal is to generate shares $\langle b \rangle_{p_i}$ in all of these systems where $b \in \{0, 1\}$. Our method makes no restriction on the size of the primes p_i, nor the underlying SPDZ-like MPC engine, thus our method can be used as a replacement for the daBit methods in [3,24] as well.

We define p_{min} to be $\mathsf{min}(p_1, \ldots, p_t)$ and we let γ be the smallest integer such that $p_{\mathsf{min}}^{\gamma} > 2^{\mathsf{sec}}$, where sec is our security parameter. For efficiency we will generate these shared bits in batches of m at a time. We define an auxiliary prime number p which satisfies

$$p > (m + \gamma \cdot \mathsf{sec}) \cdot 2^{\mathsf{sec}}.$$

[3] If the underlying MPC system is SPDZ based then a more efficient way to perform the method is using the FHE pre-processing instead of directly within the Offline phase as implied by the given protocol.

The prime p can be the same as one of the primes p_i above. All we require is that the MPC functionality $\mathcal{F}_{\mathsf{MPC}}^p$ is extended by a command which we model via the ideal functionality $\mathcal{F}_{\mathsf{MPC}}^p.\mathsf{GenBit}()$ given in Figure 5. A protocol for BDOZ and MASCOT-SPDZ for $\mathcal{F}_{\mathsf{MPC}}^p.\mathsf{GenBit}()$ is given in Figure 4, with the equivalent SPDZ protocol satisfying the same ideal functionality, see for example [10].

Functionality $\mathcal{F}_{\mathsf{MPC}}^p.\mathsf{GenBit}()$

1. For each corrupt party P_i, the functionality waits for inputs $b_i \in \mathbb{F}_p$.
2. The functionality waits for a message abort or ok from the adversary. If the message is ok then it continues.
3. The functionality then samples a bit $b \in \{0, 1\}$ and the completes the sharing to $b = \sum b_i$ by selecting shares for the honest parties.
4. The (authenticated) shares are passed to the honest players.
5. The bit b is stored in the functionality $\mathcal{F}_{\mathsf{MPC}}^p$.

Figure 5. The ideal functionality for single random bits

Our protocol will make use of the following result

Lemma 3.1. *Let $x_i \in [0, \ldots, p)$ be such that*

$$x_1 + \ldots + x_n = \begin{cases} k \cdot p, & or \\ k \cdot p + 1. \end{cases}$$

set $\Delta = \lceil p/n \rceil$ and write $x_i = l_i + \Delta \cdot h_i$ with $0 \leq l_i < \Delta$, then

$$k = \left\lceil \frac{\Delta \cdot \sum h_i}{p} \right\rceil.$$

with probability at least $1 - 1/p$.

Proof. We have, for $\epsilon \in \{0, 1\}$,

$$k = \frac{\Delta \cdot \sum h_i}{p} + \frac{\sum l_i}{p} - \frac{\epsilon}{p}.$$

We have $0 \leq \sum l_i < p$ by construction, and so the equality on k will follow as long as $\sum l_i \geq \epsilon$. But this always happens unless $\epsilon = 1$ and $\sum l_i = 0$, which happens with probability $1/p$. □

In Figure 6 we explain our protocol $\Pi_{\mathsf{RandomBit}}$ for producing shared random bits in the two MPC systems. Intuitively the protocol works as follows. The parties generate $M + \gamma \cdot \mathsf{sec}$ shared random bits in the MPC engine $\mathcal{F}_{\mathsf{MPC}}^p$ using the command GenBit. They then determine the associated k value for each shared bit using Lemma 3.1, this does not reveal any information about the hidden bit, but clearly reveals some (unimportant) information about the sharing[4]. Thinking of the sharing now as over the integers, and then reducing modulo p_i, Player P_1 can adjust his sharing so that the bit is correctly shared modulo p_i. These shares are then input into the MPC engines $\mathcal{F}_{\mathsf{MPC}}^{p_i}$. Assuming all parties are honest we now have a valid sharing.

```
Protocol $\Pi_{\mathsf{RandomBit}}$

1. Set $\Delta = \lceil p/n \rceil$.
2. For $i = 1, \ldots, m + \gamma \cdot \mathsf{sec}$ do
   (a) $\langle b_i \rangle_p \leftarrow \mathcal{F}_{\mathsf{MPC}}^{p}.\mathsf{GenBit}()$.
   (b) Let $b_{i,j}$ denote Player $j$'s value such that $\sum b_{i,j} = b_i \pmod{p}$.
   (c) Player $P_j$ writes $b_{i,j} = l_{i,j} + \Delta \cdot h_{i,j}$ with $0 \le l_{i,j} < \Delta$.
   (d) Player $P_j$ sends $h_{i,j}$ to Player $P_1$.
   (e) Player $P_1$ sets
           $$k_i = \left\lceil \frac{\Delta \cdot \sum_j h_{i,j}}{p} \right\rceil.$$
   (f) For $j = 0, \ldots, t-1$ do
       - Party $P_1$ calls $\langle b_i^{(1)} \rangle_{p_j} \leftarrow \mathcal{F}_{\mathsf{MPC}}^{p_j}.\mathsf{Input}(b_{i,1} - k_i \cdot p_1 \pmod{p_j})$.
       - Party $P_\ell$, for $\ell \neq 1$, calls $\langle b_i^{(\ell)} \rangle_{p_j} \leftarrow \mathcal{F}_{\mathsf{MPC}}^{p_j}.\mathsf{Input}(b_{i,\ell} \pmod{p_j})$.
       - The parties compute $\langle b_i \rangle_{p_j} = \sum_{\ell=1}^{n} \langle b_i^{(\ell)} \rangle_{p_j}$.
3. The parties initialize an instance of the functionality $\mathcal{F}_{\mathsf{Rand}}^{2^{\mathsf{sec}}}$. [Implementation
   note this needs to be done after the previous step so the parties have no prior
   knowledge of the output].
4. For $j = 1, \ldots, \gamma$ do
   (a) For $i = 1, \ldots, m + \gamma \cdot \mathsf{sec}$ generate $r_{i,j} \leftarrow \mathcal{F}_{\mathsf{Rand}}^{2^{\mathsf{sec}}}(m + \gamma \cdot \mathsf{sec})$.
   (b) Compute for $v = 0, \ldots, t-1$ the sharings $\langle S_{j,v} \rangle_{p_v} = \sum_i r_{i,j} \cdot \langle b_i \rangle_{p_v}$.
   (c) Compute $\langle S_j \rangle_p = \sum_i r_{i,j} \cdot \langle b_i \rangle_p$.
   (d) $S_{j,v} \leftarrow \mathsf{Open}(\langle S_{j,v} \rangle_{p_v})$ for $j = 0, \ldots, t-1$
   (e) $S_j \leftarrow \mathsf{Open}(\langle S_j \rangle_p)$.
   (f) Abort if $S_j \pmod{p_v} \neq S_{j,v}$ for any $v = 0, \ldots, t-1$.
5. Output $\langle b_i \rangle_{p_j}$ for $i = 1, \ldots, m$ and $j = 0, \ldots, t-1$.
```

Figure 6. Method to produce m shared random bits in $\mathcal{F}_{\mathsf{MPC}}^{p_0}, \ldots, \mathcal{F}_{\mathsf{MPC}}^{p_{t-1}}$

To cope with dishonest parties we check the parties are honest by verifying random linear combinations. Here we note that the initial sharing in $\mathcal{F}_{\mathsf{MPC}}^{p}$ is guaranteed to be a sharing of a bit due to the active security of the operation GenBit in $\mathcal{F}_{\mathsf{MPC}}^{p}$. Opening a random linear combination S of the shared bits in $\mathcal{F}_{\mathsf{MPC}}^{p}$ is then a subset-sum over the integers, due to the lower bound on p. We then compare this to the associated sum modulo p_i obtained from $\mathcal{F}_{\mathsf{MPC}}^{p_i}$. This has to be repeated γ times to cope with the smallest value of p_i. We thus obtain an instance of the Multiple-Subset-Sum-Problem (MSSP) considered in [22].

[4] In our security proof this information can be perfectly simulated by the simulator, and leaks no information about the actual shared value.

The protocol $\mathcal{F}^p_{\mathsf{MPC}}.\mathsf{GenBit}()$ in Figure 4 requires one secure multiplication and two rounds of communication (as a multiplication also requires a round of communication). To execute the rest of $\Pi_{\mathsf{RandomBit}}$ requires four rounds of communication (one for the initial opening to P_1, one for input into the MPC engines, one for executing $\mathcal{F}^{2^{\mathsf{sec}}}_{\mathsf{Rand}}$ and one for the final opening). If the $m + \gamma \cdot \mathsf{sec}$ bits required in $\Pi_{\mathsf{RandomBit}}$ are produced in parallel, as well as the various input/open operations etc, this means that protocol $\Pi_{\mathsf{RandomBit}}$ requires

$$m + \gamma \cdot \mathsf{sec}$$

secure multiplications in $\mathcal{F}^p_{\mathsf{MPC}}$ and $2 + 4 = 6$ rounds of communication.

3.1 Multiple Subset Sum Problem

Definition 3.1 (Multiple Subset Sum Problem [22]). *The MSSP is the problem of given weights $a_{i,j} \in \mathbb{Z}_{>0}$ for $i = 1, \ldots, k$ and $j = 1, \ldots, n$ and target values $s_1, \ldots, s_k \in \mathbb{Z}$ to find values $x_i \in \{0, 1\}$ such that*

$$\sum_{j=1}^{n} a_{i,j} \cdot x_j = s_i \quad \textit{for } i = 1, \ldots, k.$$

Just as the single subset-sum problem has a notion of density, for which one can trivially find solutions, the MSSP also has a notion of density. We define the density of an MSSP to be

$$d = \frac{n}{k \cdot \max \log a_{i,j}}.$$

We then have

Lemma 3.2 ([22]). *If $d < 0.9408$ then the MSSP problem can 'almost always' be solved with a single call to a lattice oracle.*

In this work we restrict to MSSP problems with high density, i.e. $d > 1$. In our protocol even if we set $m = 1$ the density of the subset sums S over the integers, which are revealed, is given by

$$d = \frac{1 + \gamma \cdot \mathrm{sec}}{\gamma \cdot \mathsf{sec}} > 1.$$

Informally, we note that the security of the protocol follows from the security of the underlying MPC engines, except for the leaked information. Thus we need to argue that the leaked information reveals no information about the underlying honest players' data values, and that even in the presence of malicious players the output is correct (i.e. m shared bits are the same in all $\mathcal{F}^{p_j}_{\mathsf{MPC}}$). The main potential leakage of information comes from the opened subset sum values over the integers, i.e. S. To deal with this leaked information we consider the following variant of the subset sum problem.

Definition 3.2 (Multiple Subset-Sum Guessing Problem (MSSG Problem)). *Given a set of random weights* $w_{i,j} \in [0, \ldots, 2^{\gamma \cdot \mathsf{sec}})$ *for* $i = 1, \ldots, v$ *with* $v = \gamma \cdot \mathsf{sec} + 1$ *and* $j = 1, \ldots, \gamma$*, define* $m_j = \mathsf{min}_i w_{i,j}$ *and* $s_j = \sum_i w_{i,j}$*. The problem is to distinguish between the two different distributions:*

1. *In the first distribution the challenger picks random bits* $b_i \in \{0, 1\}$ *and sets* $S_j = \sum_i b_i \cdot w_{i,j}$*. The values* $(S_1, \ldots, S_\gamma)$ *are returned to the adversary. We write this as* $\{S_j\} \leftarrow D_1$*.*
2. *In the second distribution the challenger samples values* $S_j \in [m_j, \ldots, s_j]$ *uniformly at random and returns it to the adversary. We write this as* $\{S_j\} \leftarrow D_2$*.*

If $\mathcal{A}$ *is an adversary then we define the advantage of* $\mathcal{A}$ *in solving this problem by*

$$\mathrm{Adv}_{\mathcal{A}} = 2 \cdot \Bigg| \Pr\Bigg[\ \mathcal{A}(\ \{w_{i,j}\},\ \{S_j\}\ ) = b \mid b \leftarrow \{1, 2\}, \\ w_{i,j} \leftarrow w_{i,j} \in [0, \ldots, 2^{\gamma \cdot \mathsf{sec}}), \\ S_j \leftarrow D_b \Bigg] - 1/2\ \Bigg|.$$

We say the problem is hard if $\mathrm{Adv}_{\mathcal{A}}$ *is a negligible function of* sec *for all polynomial time adversaries* $\mathcal{A}$*.*

We first discuss this problem. The condition $v = \gamma \cdot \mathsf{sec} + 1$ implies that the MSSP in the first distribution is *not* a low density multiple subset sum, and thus (if sec is chosen large enough) the underlying subset sum problem is hard.

There are approximately $(v - 1) \cdot 2^{\gamma \cdot \mathsf{sec}}$ elements in the range $[m_j, \ldots, s_j]$, but only at most $2^v = 2^{\gamma \cdot \mathsf{sec} + 1}$ of these correspond to valid subset sums. Thus the probability, in the second distribution, that a random value $S_j \in [m_j, \ldots, s_j]$ corresponds to a valid subset sum is bounded above by

$$\frac{2^{\gamma \cdot \mathsf{sec} + 1}}{(v - 1) \cdot 2^{\gamma \cdot \mathsf{sec}}} \approx \frac{2}{v - 1} = \frac{2}{\gamma \cdot \mathsf{sec}}.$$

If we set $\mathsf{sec} = 128$ then there is a $1/(\gamma \cdot 64)^\gamma$ chance that an instance selected by the second distribution is one which can be selected by the first distribution. This means that the advantage is not necessarily bounded by one for a perfect adversary.

To see this consider an adversary against the above problem, which outputs 1 if they believe the first distribution is what was sampled from and 2 if they believe it is the second. If they always output the first distribution then they are correct with probability 1/2, if they always output the second distribution then they are also correct with probability 1/2. Thus zero advantage still corresponds to an adversary which makes no intelligent choices at all.

However, now consider an adversary which outputs one with probability one if the input is from D_1, and they output two with probability one if the input is from D_2. Then they have a problem when they deduce distribution D_1 since this distribution could have arisen by chance from a challenger selection of $b = 2$ probability $1/64^\gamma$. Thus for this adversary (supposedly perfect adversary) we have

$$\begin{aligned}\mathrm{Adv}_A &= 2 \cdot \left(\frac{1}{2} \cdot (\ \Pr[\ \mathcal{A}(\ldots) = 1 \mid b = 1\ ] + \Pr[\ \mathcal{A}(\ldots) = 2 \mid b = 2\ ]\ ) - \frac{1}{2} \right) \\ &= (1 + (1 - 1/64^\gamma) - 1) = 1 - \frac{1}{64^\gamma}.\end{aligned}$$

We finally note that the best algorithms for the subset-sum problem, on elements of size $V = 2^{\mathsf{sec}}$, have time either $O(2^{\mathsf{sec}/2})$ [16] or $O(\mathsf{sec} \cdot 2^{\mathsf{sec}})$ [23], or $O(V \cdot \sqrt{\mathsf{sec}})$ [20]. With our parameters, these all have exponential time as $V = 2^{\mathsf{sec}}$.

3.2 Security of $\Pi_{\mathsf{RandomBit}}$

We now discuss the security of this protocol. Formally we want the protocol to extend the pair of functionalities $(\mathcal{F}^{p_0}_{\mathsf{MPC}}, \ldots, \mathcal{F}^{p_t}_{\mathsf{MPC}})$, with the procedure defined in Figure 7. The functionality $\mathcal{F}_{\mathsf{RandomBits}}$ internally generates a set of m secure bits, and we require that even if some information about these bits is leaked to the adversary that the remaining bits are still secure. We need this additional leaking of a subset of bits, as we do not know how the secure bits will be used in any following MPC protocol, thus we must assume the worst that a subset leaks. Even if $m - 1$ bits are revealed or some information about them is leaked then we want the final remaining bit to still be secret.

Functionality $\mathcal{F}_{\mathsf{RandomBits}}$

1. For $i = 1, \ldots, m + \gamma \cdot \mathsf{sec}$ the functionality calls $\mathcal{F}^{p}_{\mathsf{MPC}}.\mathsf{GenBit}()$ so as to store a bit b_i.
2. The bits b_i are retrieved from $\mathcal{F}^{p}_{\mathsf{MPC}}$ and are enterred into the functionalities $\mathcal{F}^{p_j}_{\mathsf{MPC}}$ for $j = 0, \ldots, t - 1$.
3. The functionality waits for a message abort or ok from the adversary. If the message is ok then it continues.
4. The functionality waits for a proper subset $\mathcal{S}$ of $\{1, \ldots, m\}$ from the adversary and returns the bits b_j for $j \in \mathcal{S}$ to the adversary. [We note this subset can be adaptively chosen, but to simplify the exposition we assume it is presented to the functionality in one go.]

Figure 7. The ideal functionality for random bits

Theorem 3.1. *Assume the problem* Multiple Subset Sum Guessing *(MSSG Problem) is hard then protocol* $\Pi_{\mathsf{RandomBit}}$ *securely implements* $\mathcal{F}_{\mathsf{RandomBits}}$ *in the* $\mathcal{F}^{2^{\mathsf{sec}}}_{\mathsf{Rand}}, \mathcal{F}^{\cdot}_{\mathsf{MPC}}\mathsf{GenBit}$*-hybrid model.*

Proof. The values S_j are produced with no wrap-around modulo p, due to the bound on M. Thus the γ values S_j define subset sums over the integers

$$S_j = \sum_{i=1}^{M+\gamma\cdot\mathsf{sec}} b_i \cdot r_{i,j}$$

where we are guaranteed that $b_i \in \{0, 1\}$.

We then check these sums against the equivalent sums modulo p_v for $v = 0, \ldots, t-1$. Since the random coefficients $r_{i,j}$ are revealed only after the parties input their shares $b_i^{(p)}$ of the bits modulo p_v, the fact that the values $S_{j,v}$ are equal to $S_j \pmod{p_v}$ implies with overwhelming probability (since $p_{\mathsf{min}}^{\gamma} > 2^{\mathsf{sec}}$) that the shared values entered modulo p_j are equal to the shared values in $\mathcal{F}_{\mathsf{MPC}}^{p}$.

Since $\mathcal{F}_{\mathsf{MPC}}^{p}.\mathsf{GenBit}$ generates values which are gauranteed to be bits, this implies the values modulo p_j are also bits, and equal to each other.

To complete the proof we must show that the protocol can be simulated. There are two sets of values which need simulating. The $h_{i,j}$ values which are sent to party P_1 by Player P_j and the subset sum values S_j and $S_{j,v}$ which are opened. We first deal with the $h_{i,j}$ values.

The simulator already knows the shares $h_{i,j}$ which the adversary should be sending to them (from $\mathcal{F}_{\mathsf{MPC}}^{p}.\mathsf{GenBit}$), and so knows whether to abort the functionality at this point. To generate the $h_{i,j}$ values for the honest parties the simulator simply picks random values for the honest parties shares $b_{i,j}$ so that they sum to zero modulo p. The honest $h_{i,j}$ are then derived from these values. These top bits of the honest shares will be a valid simulation even if the shared bit is actually one.

To simulate the value S_j, and hence $S_{j,v}$, we simply define the trivial simulator. The values $r_{i,j}$ are sampled using the ideal functionality $\mathcal{F}_{\mathsf{Rand}}^{2^{\mathsf{sec}}}$, and then the values $m_j = \mathsf{min}\ r_{i,j}$ and $s_j = \sum r_{i,j}$ are computed. It then picks a random value $S_j \in [m_j, \ldots, s_j]$ and returns S_j and $S_{j,v} = S_j \pmod{p_v}$, for every v, to the adversary. When the adversary selects a proper subset $\mathcal{S}$ of $\{1, \ldots, m\}$ to be opened, it queries the ideal functionality and returns these values.

We now show that an environment that can distinguish between a real-world execution and the ideal-world execution with this simulator can be used to solve an arbitrary instance of the MSSG problem. We let $\mathcal{A}$ denote our adversary against the MSSG problem. This takes as input $t = \mathsf{sec} + 1$ values $w_{1,j}, \ldots, w_{t,j}$, for $j = 1, \ldots, \gamma$, and a target sum T_j for $j = 1, \ldots, \gamma$. The adversary $\mathcal{A}$ runs the environment twice as follows, where we let the run be denoted by the variable $r \in \{1, 2\}$.

1. The adversary $\mathcal{A}$ selects an index $i_r^* \in \{1, \ldots, m\}$.
2. The adversary sets $r_{i_r^*,j}^{(r)}$ to be $w_{t,j}$.
3. The adversary sets $r_{i+m,j}^{(r)} = w_{i,j}$ for $i = 1, \ldots, \mathsf{sec}$.
4. The adversary selects all other values $r_{i,j}^{(r)}$, for run r, at random from $[0, \ldots, 2^{\mathsf{sec}})$.
5. For $i \in \{1, \ldots, m\} \setminus \{i_r^*\}$ the adversary selects a bit $b_i^{(r)} \in \{0, 1\}$.

6. The adversary sets $S_j(r) = T_j + \sum b_i^{(r)} \cdot r_{i,j}^{(r)}$, where the sum is over $i \in \{1, \ldots, m\} \setminus \{i_r^*\}$.
7. The adversary then runs the environment, revealing the values $S_j^{(r)}$, and $S_j^{(r)} \pmod{p_v}$ as required.
8. At some point the environment in run r will request a subset of queries $\mathcal{S}_r \subset \{1, \ldots, M\}$. At this point for every $i \in \mathcal{S}_r$ the adversary returns $b_i^{(r)}$ if $i \neq i_r^*$. If $i = i_r^*$ then the adversary returns $r - 1$, i.e. 0 in the first run and 1 in the second run.
9. Finally the environmental distinguisher will return its result of *real* or *ideal*.
10. In the case of a value returned of *real* in either run, the adversary $\mathcal{A}$ decides its input problem was from the first MSSG distribution.
11. Otherwise the adversary decides that its input was from the second MSSG distribution.

We now discuss why this adversary solves the MSSG problem. Notice that if $i_r^* \notin \mathcal{S}_r$ for one of the two executions, then this execution is indistinguishable by definition to the real world execution when the input problem is from the first MSSG distribution and is equal to the ideal world execution when the input problem is from the second MSSG distribution.

When $i_r^* \in \mathcal{S}_r$, and the input problem is from the first MSSG distribution, and the bit corresponding to this solution is correct (i.e. is actually equal to $r - 1$) then the adversary is presenting an execution indistinguishable from the real world. If the bit is wrong, i.e. not equal to $r-1$, then the execution is unlikely to be identical to a real execution (unless there is a solution to the subset sum with $b_{i_r^*}^{(r)}$ equal to $2 - r$). Since we run the adversary twice in such a case we are guaranteed that if the input is from the first MSSG distribution one of our executions will be correctly equal to the real distribution.

In the case when $i_r^* \in \mathcal{S}_r$ for both values of r and the input distribution to $\mathcal{A}$ is from the second MSSG distribution, then the view given to the distinguisher is that of the simulated distribution.

Thus in all cases any distinguisher which can distinguish between the real and ideal world will be able to be used by $\mathcal{A}$ to solve the MSSG problem. □

As pointed out to us by Carsten Baum we can remove the need to assume our variant of the subset-sum protocol in the above theorem if we increase the number size of our subset sum from $m + \gamma \cdot \mathsf{sec}$ to $m + 2 \cdot \gamma \cdot \mathsf{sec}$. This obviously reduces the efficieny of the protocol. If one is willing to pay this price then security becomes a purely statistical argument. Such an argument is similar to arguments used in prior works in various situations, see for example [18,25].

4 Sampling Distributions In MPC

In this section we explain how we perform the various sampling operations we need in the key generation algorithm. We assume that two actively-secure (with-abort) MPC functionalities, as in Figure 2, have already been initiated, one for the prime p_0 and one for the prime p_1. We call these two functionalities $\mathcal{F}_{\mathsf{MPC}}^{p_0}$ and $\mathcal{F}_{\mathsf{MPC}}^{p_1}$. As explained earlier these can be instantiated with either BDOZ or MASCOT-SPDZ. Using these functionalities we instantiate the protocol $\Pi_{\mathsf{RandomBit}}$, from Section 3, to produce doubly-authenticated bits in both $\mathcal{F}_{\mathsf{MPC}}^{p_0}$ and $\mathcal{F}_{\mathsf{MPC}}^{p_1}$.

4.1 Protocol Π_{Hamming}

To sample our secret key we will need to sample a shared vector of length N, with elements in $\{-1, 0, 1\}$, with approximate Hamming weight h, i.e. we sample from $B(h/N)^N$. The values of N and h are selected such they are both powers of two, so we set $N = 2^\nu$ and $h = 2^\ell$. The reader should think of values of $N = 32768$ and $h = 64$, which are used in `SCALE-MAMBA`'s default configuration. To sample such vectors we use protocol $\Pi_{\mathsf{Hamming}}(h, N)$ given in Figure 8.

Protocol $\Pi_{\mathsf{Hamming}}(h, N)$

```
1. Set $N = 2^\nu$ and $h = 2^\ell$.
2. Using $\Pi_{\mathsf{RandomBit}}$ generate $H = N \cdot (\nu - \ell)$ shared random bits $\langle b_i^j \rangle_{p_0}$ and $\langle b_i^j \rangle_{p_1}$,
   for $i \in \{1, \ldots, N\}$ and $j \in \{1, \ldots, \nu - \ell\}$.
3. For $i \in \{1, \ldots, N\}$
   (a) $\langle b_i \rangle_{p_0} \leftarrow \langle b_i^1 \rangle_{p_0} \cdot \ldots \cdot \langle b_i^{\nu-\ell} \rangle_{p_0}$.
   (b) $\langle b_i \rangle_{p_1} \leftarrow \langle b_i^1 \rangle_{p_1} \cdot \ldots \cdot \langle b_i^{\nu-\ell} \rangle_{p_1}$.
4. Using $\Pi_{\mathsf{RandomBit}}$ generate $N$ shared random bits $\langle s_i \rangle_{p_0}$ and $\langle s_i \rangle_{p_1}$, for $i \in$
   $\{1, \ldots, N\}$.
5. For $i \in \{1, \ldots, N\}$
   (a) $\langle b_i \rangle_{p_0} \leftarrow \langle b_i \rangle_{p_0} \cdot (2 \cdot \langle s_i \rangle_{p_0} - 1)$.
   (b) $\langle b_i \rangle_{p_1} \leftarrow \langle b_i \rangle_{p_1} \cdot (2 \cdot \langle s_i \rangle_{p_1} - 1)$.
6. Return, for $i \in \{1, \ldots, N\}$, the shares $\langle b_i \rangle_{p_0}$ and $\langle b_i \rangle_{p_1}$.
```

Figure 8. Method to produce vectors of (expected) Hamming weight h with elements in $\{-1, 0, 1\}$

We again estimate the cost of this operation. If we assume that the two calls to $\Pi_{\mathsf{RandomBit}}$ require only one batch operation then these operation requires 6 rounds of communication, $(H + N + \mathsf{sec})$ multiplications in $\mathcal{F}_{\mathsf{MPC}}^{p}$[5]. The products to produce the initial bits b_i require $\lceil \log_2(\nu - \ell) \rceil$ rounds of communication and $(\nu - \ell - 1) \cdot N$ multiplications in both $\mathcal{F}_{\mathsf{MPC}}^{p_0}$ and $\mathcal{F}_{\mathsf{MPC}}^{p_1}$. Whereas the products to produce the final signed bits require one round of communication and N multiplications in both $\mathcal{F}_{\mathsf{MPC}}^{p_0}$ and $\mathcal{F}_{\mathsf{MPC}}^{p_1}$. Hence in total we require

$$6 + \lceil \log_2(\nu - \ell) \rceil + 1 = 7 + \lceil \log_2(\nu - \ell) \rceil$$

rounds of communication and

$$N_{\mathsf{Hamming},0} = N + H + \mathsf{sec} + (\nu - \ell + 1) \cdot N,$$

[5] Note that $\gamma = 1$ since p_0, p_1 are both big.

$$N_{\mathsf{Hamming},1} = (\nu - \ell + 1) \cdot N$$

multiplications in total $\Pi^{p_0}_{\mathsf{MPC}}$ and $\Pi^{p_1}_{\mathsf{MPC}}$ respectively.

4.2 Protocol Π_{Binomial}

The protocol for sampling shared values from the distribution $\mathsf{dN}(\sigma^2, N)$ is relatively straightforward, and is given in Figure 9. Apart from the generation of the random bits this protocol consists entirely of linear operations. Thus the round complexity is six and it requies $N_{\mathsf{Binomial}} = 2 \cdot k \cdot N + \mathsf{sec}$ multiplications in $\mathcal{F}^p_{\mathsf{MPC}}$.

```
Protocol $\Pi_{\mathsf{Binomial}}(\sigma^2, N)$

1. Define $k$ by $\sigma = \sqrt{k/2}$.
2. Using $\Pi_{\mathsf{RandomBit}}$ generate $2 \cdot k \cdot N$ shared random bits $\langle b_i^j \rangle_{p_0}$ and $\langle b_i^j \rangle_{p_1}$, for
   $i \in \{1, \ldots, N\}$ and $j \in \{0, \ldots, 2 \cdot k - 1\}$.
3. For $i \in \{1, \ldots, N\}$
   (a) $\langle b_i \rangle_{p_0} \leftarrow \sum_{j=0}^{k-1} \langle b_i^{2 \cdot j} \rangle_{p_0} - \langle b_i^{2 \cdot j+1} \rangle_{p_0}$.
   (b) $\langle b_i \rangle_{p_1} \leftarrow \sum_{j=0}^{k-1} \langle b_i^{2 \cdot j} \rangle_{p_1} - \langle b_i^{2 \cdot j+1} \rangle_{p_1}$.
4. Return, for $i \in \{1, \ldots, N\}$, the shares $\langle b_i \rangle_{p_0}$ and $\langle b_i \rangle_{p_1}$.
```

Figure 9. Method to produce elements from $\mathsf{dN}(\sigma^2, N)$

4.3 Protocol Π_{Uniform}

Our final protocol is a rather trivial one, it allows the parties to sample a uniform element from $\mathbb{Z}_q$ in a secret shared form, we give it in Figure 10.

```
Protocol $\Pi_{\mathsf{Uniform}}(N)$

1. For $i \in \{1, \ldots, M\}$
   (a) For $j \in \{1, \ldots, n\}$
       i. Player $P_i$ selects $u_i^j \in \mathbb{F}_{p_0}$, and $v_i^j \in \mathbb{F}_{p_1}$,
       ii. Execute $\langle u_i^j \rangle_{p_0} \leftarrow \mathsf{Input}(P_i)$.
       iii. Execute $\langle v_i^j \rangle_{p_1} \leftarrow \mathsf{Input}(P_i)$.
   (b) $\langle u_i \rangle_{p_0} \leftarrow \langle u_i^1 \rangle_{p_0} + \ldots + \langle u_i^n \rangle_{p_0}$.
   (c) $\langle v_i \rangle_{p_0} \leftarrow \langle v_i^1 \rangle_{p_0} + \ldots + \langle v_i^n \rangle_{p_0}$.
2. Return, for $i \in \{1, \ldots, N\}$, the shares $\langle u_i \rangle_{p_0}$ and $\langle v_i \rangle_{p_1}$.
```

Figure 10. Protocol to sample a uniform element from $\mathbb{Z}_q$

5 SPDZ KeyGeneration

Given the previous algorithms to generate various distributions the computation of the actual key generation algorithm becomes relatively straight forward. We first sample the various distributions modulo p_0 and p_1, then we produce the square of the secret key (needed for the key switching matrices), and then we output the public key and recombine it using the CRT, then we do the same to each players component of the secret key. The overall protocol is given in Figure 11; note that in line 3 one can select as to whether to choose the secret key from a restricted Hamming weight or from the centred binomial distribution. To make the protocol easier to follow we use the notation $\langle \mathbf{a} \rangle_{p_0}$ etc to denote a vector of N shares modulo p_0, i.e. $\langle \mathbf{a} \rangle_{p_0} = (\langle a_0 \rangle_{p_0}, \ldots, \langle a_{N-1} \rangle_{p_0})$.

We let $\langle \mathbf{a} \rangle_{p_0} \odot \langle \mathbf{b} \rangle_{p_0}$ denote the multiplication of two such vectors when considered as elements in the ring R_{p_0}. This requires one round of communication and N^2 secure multiplications (or $N \cdot (N-1)/2$ secure multiplications if $\mathbf{a} = \mathbf{b}$). If one vector is in the clear then we write $\mathbf{a} \odot \langle \mathbf{b} \rangle_{p_0}$, which is a linear operation and hence for "free". A more efficient method to multiply is to use the FFT algorithm, which recall is a linear operation and thus 'free' when executed in the MPC engine. To multiply using FFT we utilize

$$\langle \mathbf{a} \rangle_{p_0} \odot \langle \mathbf{b} \rangle_{p_0} = \mathsf{FFT}^{-1}(\mathsf{FFT}(\langle \mathbf{a} \rangle_{p_0}) \cdot \mathsf{FFT}(\langle \mathbf{b} \rangle_{p_0})$$

which requires only N secure multiplications and one round of communication.

We now examine each of the operations in this algorithm in turn. The lines 1-6 can all be executed in parallel and so require

$$\max\Big(6 + \lceil \log_2(\nu - \ell) \rceil + 1, 6, 1\Big) = 6 + \lceil \log_2(\nu - \ell) \rceil$$

rounds of communication [6] in the case where we select restricted Hamming weight secret key and

$$\max\Big(6, 6, 1\Big) = 6$$

for the case of the secret key generated from a centred binomial distribution. The number of secure multiplications is given by, in the two cases,

$$N_{\mathsf{Hamming},0} + N_{\mathsf{Hamming},1} + 2 \cdot N_{\mathsf{Binomial}} = N + H + \mathsf{sec} + 2 \cdot (\nu - \ell + 2 \cdot k - 1) \cdot N,$$
$$3 \cdot N_{\mathsf{Binomial}} = 6 \cdot k \cdot N.$$

Lines 7 and 9 are linear operations and thus can be executed as purely local operations. Line 8 requires one round of communication and N multiplications in $\mathcal{F}_{\mathsf{MPC}}^{p_0}$. Note, in lines 7-9 we only have to compute $\mathfrak{sk}^2$ modulo p_0 as it is multiplied by p_1 when added into $\mathbf{b}_{\mathfrak{sk},\mathfrak{sk}^2}$. Lines 10-13 can also be performed in parallel and hence require only one round of communication. The lines 16-21 are all local operations, and hence are for "free". Lines 25-24 are, again, able to be

[6] Of course in practice we generate the secure bits in batches and hence this is just the minimal number of rounds required.

Protocol Π_{KeyGen}

```
 - [$Generate\ the\ various\ values\ required$]
 1. $(\langle\mathbf{a}\rangle_{p_0}, \langle\mathbf{a}\rangle_{p_1}) \leftarrow \Pi_{\mathsf{Uniform}}(N)$.
 2. $(\langle\mathbf{a}'\rangle_{p_0}, \langle\mathbf{a}'\rangle_{p_1}) \leftarrow \Pi_{\mathsf{Uniform}}(N)$.
 3. $(\langle\mathbf{s}\rangle_{p_0}, \langle\mathbf{s}\rangle_{p_1}) \leftarrow \Pi_{\mathsf{Hamming}}(h, N)$ or $(\langle\mathbf{s}\rangle_{p_0}, \langle\mathbf{s}\rangle_{p_1}) \leftarrow \Pi_{\mathsf{Binomial}}(N)$.
 4. $(\langle\mathbf{e}\rangle_{p_0}, \langle\mathbf{e}\rangle_{p_1}) \leftarrow \Pi_{\mathsf{Binomial}}(N)$.
 5. $(\langle\mathbf{e}'\rangle_{p_0}, \langle\mathbf{e}'\rangle_{p_1}) \leftarrow \Pi_{\mathsf{Binomial}}(N)$.
 6. For $i \in \{2, \ldots, n\}$
    (a) $\langle\mathfrak{sk}_{0,i}\rangle_{p_0} \leftarrow \mathsf{Input}(P_i)$ for a random vector $\mathfrak{sk}_{0,i} \in R_{p_0}$ selected by player $P_i$.
    (b) $\langle\mathfrak{sk}_{1,i}\rangle_{p_1} \leftarrow \mathsf{Input}(P_i)$ for a random vector $\mathfrak{sk}_{1,i} \in R_{p_1}$ selected by player $P_i$.
 - [$Compute\ the\ square\ of\ the\ secret\ key$]
 7. $\langle\mathbf{f}\rangle_{p_0} \leftarrow \mathsf{FFT}(\langle\mathbf{s}\rangle_{p_0})$.
 8. $\langle\mathbf{f}'\rangle_{p_0} \leftarrow \langle\mathbf{f}\rangle_{p_0} \cdot \langle\mathbf{f}\rangle_{p_0}$ [This is the component wise product].
 9. $\langle\mathbf{s}'\rangle_{p_0} \leftarrow \mathsf{FFT}^{-1}(\langle\mathbf{f}'\rangle_{p_0})$.
 - [$Open\ the\ values$ $\mathbf{a}$ $and$ $\mathbf{a}'$]
10. $\mathbf{a}_0 \leftarrow \mathsf{Open}(\langle\mathbf{a}\rangle_{p_0})$.
11. $\mathbf{a}_1 \leftarrow \mathsf{Open}(\langle\mathbf{a}\rangle_{p_1})$.
12. $\mathbf{a}'_0 \leftarrow \mathsf{Open}(\langle\mathbf{a}'\rangle_{p_0})$.
13. $\mathbf{a}'_1 \leftarrow \mathsf{Open}(\langle\mathbf{a}'\rangle_{p_1})$.
14. $\mathbf{a} \leftarrow \mathsf{CRT}([\mathbf{a}_0, \mathbf{a}_1], [p_0, p_1])$.
15. $\mathbf{a}_{\mathfrak{sk},\mathfrak{sk}^2} \leftarrow \mathsf{CRT}([\mathbf{a}'_0, \mathbf{a}'_1], [p_0, p_1])$.
 - [$Compute$ $\mathbf{b}$ $and$ $\mathbf{b}'$]
16. $\langle\mathbf{b}\rangle_{p_0} \leftarrow \mathbf{a}_0 \odot \langle\mathbf{s}\rangle_{p_0} + p \cdot \langle\mathbf{e}\rangle_{p_0}$.
17. $\langle\mathbf{b}\rangle_{p_1} \leftarrow \mathbf{a}_1 \odot \langle\mathbf{s}\rangle_{p_1} + p \cdot \langle\mathbf{e}\rangle_{p_1}$.
18. $\langle\mathbf{b}'\rangle_{p_0} \leftarrow \mathbf{a}'_0 \odot \langle\mathbf{s}\rangle_{p_0} + p \cdot \langle\mathbf{e}'\rangle_{p_0} - p_1 \cdot \langle\mathbf{s}'\rangle_{p_0}$.
19. $\langle\mathbf{b}'\rangle_{p_1} \leftarrow \mathbf{a}'_1 \odot \langle\mathbf{s}\rangle_{p_1} + p \cdot \langle\mathbf{e}'\rangle_{p_1}$.
 - [$Fix\ the\ final\ key\ for\ the\ sharing\ of\ the\ secret\ key$]
20. $\langle\mathfrak{sk}_{0,1}\rangle_{p_0} \leftarrow \langle\mathbf{s}\rangle_{p_0} - \sum_{i=2}^{n} \langle\mathfrak{sk}_{0,i}\rangle_{p_0}$.
21. $\langle\mathfrak{sk}_{1,1}\rangle_{p_1} \leftarrow \langle\mathbf{s}\rangle_{p_1} - \sum_{i=2}^{n} \langle\mathfrak{sk}_{1,i}\rangle_{p_1}$.
22. For all $i \in \{1, \ldots, n\}$ assign $\mathfrak{sk}_i \leftarrow \mathsf{CRT}([\mathfrak{sk}_{0,i}, \mathfrak{sk}_{1,i}], [p_0, p_1])$.
23. $P_1 \leftarrow \mathsf{Output}(\langle\mathfrak{sk}_{0,1}\rangle_{p_0})$.
24. $P_1 \leftarrow \mathsf{Output}(\langle\mathfrak{sk}_{1,1}\rangle_{p_1})$.
 - [$Open\ the\ values$ $\mathbf{b}$ $and$ $\mathbf{b}'$]
25. $\mathbf{b}_0 \leftarrow \mathsf{Open}(\langle\mathbf{b}\rangle_{p_0})$.
26. $\mathbf{b}_1 \leftarrow \mathsf{Open}(\langle\mathbf{b}\rangle_{p_1})$.
27. $\mathbf{b}'_0 \leftarrow \mathsf{Open}(\langle\mathbf{b}'\rangle_{p_0})$.
28. $\mathbf{b}'_1 \leftarrow \mathsf{Open}(\langle\mathbf{b}'\rangle_{p_1})$.
29. $\mathbf{b} \leftarrow \mathsf{CRT}([\mathbf{b}_0, \mathbf{b}_1], [p_0, p_1])$.
30. $\mathbf{b}_{\mathfrak{sk},\mathfrak{sk}^2} \leftarrow \mathsf{CRT}([\mathbf{b}'_0, \mathbf{b}'_1], [p_0, p_1])$.
31. Output $\mathbf{a}, \mathbf{b}, \mathbf{a}_{\mathfrak{sk},\mathfrak{sk}^2}$ and $\mathbf{b}_{\mathfrak{sk},\mathfrak{sk}^2}$ to all players, and $\mathfrak{sk}_i$ to player $P_i$.
```

Figure 11. The Distributed BGV Key Generation Protocol

done in parallel and so require only one round of communication. The remaining lines are purely local organization of data into the correct format for outputing. Thus the total number of rounds of communication (assuming all shared random bits are produced in a single batch) is

$$12 + \lceil \log_2(\nu - \ell) \rceil \quad \text{or} \quad 12$$

depending on which variant one is using for the secret key.

If we take typical values of $n = 2$, $N = 32768$, $h = 64$, and $\mathsf{sec} = 128$ then these work out to be 3277056 mults in $\Pi_{\mathsf{MPC}}^{p_0}$ and 3244288 in $\Pi_{\mathsf{MPC}}^{p_1}$ and 16 rounds of communication.

Theorem 5.1. *The protocol Π_{KeyGen} UC-securely realises the functionality $\mathcal{F}_{\mathsf{KeyGen}}$ against a static, active adversary corrupting at most $n - 1$ parties in the $\mathcal{F}_{\mathsf{MPC}}^{p}$-hybrid model, assuming the decision subset-sum problem is hard.*

Proof. We define the simulator $\mathcal{S}$ as follows. The simulator emulates the behaviour of honest parties exactly, but additionally does the following:

- At the start of the execution, the simulator initialises a local copy of $\mathcal{F}_{\mathsf{MPC}}^{p_0}$ and $\mathcal{F}_{\mathsf{MPC}}^{p_1}$ and sends the message *start* to $\mathcal{F}_{\mathsf{KeyGen}}$ and awaits the public key $\mathfrak{pk} = (\bar{\mathbf{a}}, \bar{\mathbf{b}}, \bar{\mathbf{a}}_{\mathfrak{sk},\mathfrak{sk}^2}, \bar{\mathbf{b}}_{\mathfrak{sk},\mathfrak{sk}^2})$ in response.
- When the adversary and simulator execute Π_{Uniform}, the simulator replaces the values $\mathbf{a} \mod p_0$, $\mathbf{a} \mod p_1$, $\mathbf{a}' \mod p_0$ and $\mathbf{a}' \mod p_1$ stored in the instances of $\mathcal{F}_{\mathsf{MPC}}^{p_0}$ and $\mathcal{F}_{\mathsf{MPC}}^{p_1}$, respectively, with $\bar{\mathbf{a}} \mod p_0$, $\bar{\mathbf{a}} \mod p_1$, $\bar{\mathbf{a}}_{\mathfrak{sk},\mathfrak{sk}^2} \mod p_0$ and $\bar{\mathbf{a}}_{\mathfrak{sk},\mathfrak{sk}^2} \mod p_1$.
- In Step 6, for each $j \in [n]$, if j is corrupt and $j > 2$ then the simulator awaits the input $\mathfrak{sk}_{j,0}$ and $\mathfrak{sk}_{j,1}$ for each corrupt party P_j and constructs $\mathfrak{sk}_j \leftarrow \mathsf{CRT}(\mathfrak{sk}_{j,0}, \mathfrak{sk}_{j,1})$, and sends these to $\mathcal{F}_{\mathsf{KeyGen}}$.
- Just before opening $\mathbf{b}_0$, $\mathbf{b}_1$, $\mathbf{b}'_0$ and $\mathbf{b}'_1$ the simulator replaces these values stored in the instances of $\mathcal{F}_{\mathsf{MPC}}^{p_0}$ and $\mathcal{F}_{\mathsf{MPC}}^{p_1}$ with $\bar{\mathbf{b}} \mod p_0$, $\bar{\mathbf{b}} \mod p_1$, $\bar{\mathbf{b}}_{\mathfrak{sk},\mathfrak{sk}^2} \mod p_0$ and $\bar{\mathbf{b}}_{\mathfrak{sk},\mathfrak{sk}^2} \mod p_1$.

Since the only inputs to the protocol are randomly sampled by parties, the simulator can perfectly emulate the behaviour of honest parties throughout, as the environment does not observe the random tape of honest parties or the simulator. Indeed, since $\mathcal{F}_{\mathsf{MPC}}^{p_0}$ and $\mathcal{F}_{\mathsf{MPC}}^{p_1}$ are used as black boxes and the only communication between parties occurs via these functionalities, which are emulated locally honestly by the simulator, the replacements made in the simulation outlined are executed without being observed by the environment (at this point). Moreover, the inputs of corrupt parties can be extracted trivially and passed on to $\mathcal{F}_{\mathsf{KeyGen}}$ so that the final outputs have the correct distribution.

It only remains to show that the environment cannot observe a difference between the transcripts in a real execution and an execution in which the replacements described above are made. While the final outputs of the real and ideal worlds is the same, the distribution of the transcript differs since communication generated in the execution of the sampling subprotocols depends on the secret $\bar{\mathbf{s}}$ which is (implicitly) generated by the functionality $\mathcal{F}_{\mathsf{KeyGen}}$ when executing $\mathsf{KeyGen}()$ and cannot be computed by the simulator from the public key without breaking the LWE assumption for the security of the key. We must show that the amount by which the distributions differ is negligible.

The only time information stored in $\mathcal{F}_{\mathsf{MPC}}^{p_0}$ or $\mathcal{F}_{\mathsf{MPC}}^{p_1}$ is either revealed to the parties or is generated by parties is in the bits shared in $\Pi_{\mathsf{RandomBits}}$ and in Π_{Uniform}. In the protocols Π_{Binomial} and Π_{Hamming} the parties obtain bits from $\Pi_{\mathsf{RandomBits}}$, and in the remainder of the protocol Π_{KeyGen}, the correctness of the computations is guaranteed by the security of the black boxes $\mathcal{F}_{\mathsf{MPC}}^{p_0}$ and $\mathcal{F}_{\mathsf{MPC}}^{p_1}$. In Π_{Uniform}, every party contributes to every secret, and since there is at least one honest party (that samples uniformly), the output is always uniform. Thus it suffices to argue that nothing can be learnt in $\Pi_{\mathsf{RandomBits}}$ about the bits that are generated (which are later used to generate the public key) without solving the subset sum problem. But this follows from Theorem 3.1. Thus no distinguishing environment can exist, by the choice of parameters for the subset sum, and therefore Π_{KeyGen} UC-securely realises $\mathcal{F}_{\mathsf{KeyGen}}$. $\square$

6 Implementation

In our implementation we use MASCOT [19] as the base protocol used for our one-time setup phase for SPDZ. Our solution is built on top of the `SCALE-MAMBA` [2] framework, and we therefore re-used a lot of their already existing codebase. As explained in the introduction our key generation protocol, as it uses MASCOT, is inherently $O(n^2)$ in nature. This seems to be unavoidable as the only practical $O(n)$ MPC protocol known is SPDZ, which is exactly the MPC protocol we are trying to instantiate with our key generation protocol.

Selection of FHE parameters We recall that the two-leveled BGV key generation procedure requires us to choose two prime moduli p_0 and p_1 and a polynomial degree N to define the ciphertext space $R_q \cong R_{p_0} \times R_{p_1}$ and a prime p which defines the size of our plaintext space. In addition we require that the relations presented in 2.2 hold.

The precise sizes of these parameters are derived from a noise analysis of how the resulting encryption scheme is used, which takes into account the circuit being computed, the zero-knowledge proofs required, and the distributed decryption procedure, and the computational difficulty of the Ring-LWE problem. This analysis is quite involved and we referred to the `SCALE-MAMBA` documentation [2] to obtain the required parameters.

This gives us (for example) that to guarantee a computational security of 128 bits with a polynomial degree $N = 32768$, our ciphertext modulus q has to verify $q < 2^{883}$. For such parameters, the trusted setup of `SCALE-MAMBA`, as we have discussed earlier, produces a secret key with Hamming Weight exactly 64, and uses noise vectors distributed according to a centred binomial distribution with standard deviation of $3.16 = \sqrt{10}$. As a first set of experiments we use exactly the same methodology to select the secret key, but we pick a secret key with expected Hamming Weight 64. This does not change the noise analysis (which is done using an expected noise methodology for the secret key in any case), and thus we end up with the same system parameters as used in `SCALE-MAMBA`. In particular for a plaintext modulus of 128 bits this leads us to use a p_0 modulus of 345 bits and a p_1 modulus of 225 bits. So in order to run the key generation protocol presented above, we will need to run two instances of the MASCOT protocol; one for the 345 bits prime p_0 and another for the 225 bits prime p_1.

We tried different sets of parameters in our experiments, which provide BGV keys for a SPDZ modulus of 64 and 128 bits, always taking the same parameters as the Setup phase for `SCALE-MAMBA`. In Table 1 we report the prime sizes, in bits, required for each set of parameters for two and three parties.

Number of Parties, n	2				3			
FHE Plaintext Size, $\log_2 p$	64		128		64		128	
Polynomial Degree	16384		32768		16384		32768	
$p_0 \| p_1$ bit length	216	164	345	225	217	163	346	224

Table 1. Parameter Sizes.

With these size of primes, we can now set the value m for our batch size of shared bits which has to respect the bound stated in 3. In practice we want to have m a divisor of the total number of shared bits that we will need, and also small enough to avoid RAM or network overflow. Empirically we found that taking $50000 \leq m \leq 100000$ (depending on the setting) gives us good results.

Extended Random Oblivious Transfer: The `SCALE-MAMBA` framework does not have an implementation of the offline phase of MASCOT, and an implementation of the extended Random Oblivious Transfer for a prime field $\mathbb{F}_p$, thus we needed to implement these. The triple generation method of MASCOT [19][Protocol 4] makes use of an extended Correlated Oblivious Transfer (COT) protocol. For this we used the passively secure protocol of Frederiksen et al. [13][Full Version, Figure 19] (which is essentially the protocol of Ishai et al [17]). Such a passively secure COT protocol is sufficient due to how it is used in the MASCOT offline phase. This COT protocol was already implemented in `SCALE-MAMBA`. For two parties P_i and P_j with the former acting as the sender and the later as the receiver, calls to the Correlated Oblivious Transfer output values $\{M_0^k, M_1^k = M_0^k + \Delta_i\}_{k\in[n]}$ to P_i and $\{M_{\mathbf{b}_k}^k\}_{k\in[n]}$ to P_j, where $\Delta_i \in \mathbb{F}_{2^{128}}$ is the input from P_i and $\mathbf{b} \in \{0,1\}^n$ is the choice vector from P_j, and $\forall k \in [n]\, M_0^k \leftarrow_{\$} \mathbb{F}_{2^{128}}$

To obtain extended Random Oblivious Transfer (ROT), from these extended COTs, we used the decorrelation technique presented in [8][Figure 15] which consists in both parties hashing the output of the extended COT. This gives us an extended ROT in $\mathbb{F}_{2^{128}}$. However, we want to run the MASCOT protocol on prime fields $\mathbb{F}_{p_0}$ and $\mathbb{F}_{p_1}$, so to translate our extended ROT from $\mathbb{F}_{2^{128}}$ to $\mathbb{F}_p$, we take $\lceil \frac{\log_2(p)+128}{128} \rceil$ outputs in $\mathbb{F}_{2^{128}}$, concatenate them together and take the result mod p.

Results of our experiments Our implementation of the key generation protocol was tested in a LAN setting, with each party running on an Intel i7-7700K CPU with 32GB of RAM over a 10Gb/s network switch. In our experiments we found that executing more threads than the available number of cores (in our case eight) gave a performance improvement. This is because the computation between receiver and sender in the OT protocols is asymmetric, resulting in each party sometimes waiting for the other to perform some computation.

For each prime size, we report the throughput of our MASCOT implementation regarding triple generation in Table 2. We experimented with different number of threads, and for each case we give our results for 1, 5 and 10 threads. By comparison to [19][Figure 9], our implementation of the MASCOT protocol for triple generation seems to have room for improvement. In particular for a 128 bit field, we are about 7 times slower than MP-SPDZ [12]. Therefore, we believe that a better implementation of MASCOT using less CPU resources would give us a significant amelioration of our BGV key generation benchmark.

n	2				3			
$\log_2 p$	64		128		64		128	
1 Thread	126.77	177.1	70.01	133.29	58.78	79.01	28.7	57.22
5 Thread	587.77	784.69	292.18	567.62	276.38	376.65	134.62	279.27
10 Thread	759.1	1032.57	365.93	740.05	389.19	547.27	185.32	379.34

Table 2. Triple Throughput for $\mathbb{F}_{p_0}$ and $\mathbb{F}_{p_1}$ in Triples per Second

Finally in Table 3 we give figures related to the throughput of the secure randomly shared bits, namely the throughput of the algorithm in Figure 6, in the four different scenarios we experimented with. For each case, we give our result while also running the MASCOT triple generation, and also assuming that this offline phase has already been done beforehand. In the same table we eventually give the total runtime for the distributed BGV key generation procedure, which includes the time to perform the MASCOT offline phase.

We can observe that for two parties it takes between 47 minutes and five hours to complete the distributed key generation. Although it may seem inefficient, we argue that this protocol needs only to be done once in order to enable future computation of the offline phase of SPDZ. Moreover, we have shown that our protocol is highly parallelizable. So in practice, if such a protocol was to be run on high end servers owned by cloud service providers, the total runtime could be drastically reduced.

We pause to compare these run-times to the covertly secure distributed key generation protocol presented in [10]. This protocol did not produce public and

Number of Parties	2		3	
FHE Plaintext size	64	128	64	128
Shared bit throughput including offline time (b/s)	416.67	263.16	194.55	124.69
Shared bit throughput excluding offline time (b/s)	1515.15	1449.28	1149.43	1041.67
BGV Key Generation Wall Clock	47m17s	2h35m34s	1h30m24s	4h55m44s

Table 3. Shared bit throughput and total time for the KeyGen protocol.

secret keys with the same distribution as the non-distributed version. For plaintexts of 64-bits the authors of this paper report 12 and 16 seconds key generation time for $n = 2$ and $n = 3$, for a covert security of 1/10, i.e. an adversary can cheat with probability 1/10. For plaintexts of 128-bits the times are 33 and 44 seconds. The execution time of this covertly secure protocol appears linear in c, where the covert security is $1/c$. Thus to obtain comparable security to our protocol, the protocol in [10] would be utterly inpractical.

6.1 Changing the Standard Deviation

Our protocol can also be run when we select the standard deviation for the centred binomial distributions to be equal to $\sigma = \sqrt{1/2} = 0.707$. The analysis of the parameters, given in Table 1 is roughly the same. However, the associated run times for key generation become faster as we no longer need to generate as many doubly authenticated bits. This is reflected in Table 4. We see that by choosing $\sigma = 0.707$, we need only two bits instead of 40 for the sampling from the centred binomial distribution. We thus get a factor of at least 2.5 improvement over the previous setting.

Number of Parties	2		3	
FHE Plaintext size	64	128	64	128
BGV Key Generation Wall Clock	16m10s	1h03m19s	28m46s	1h52m54s

Table 4. Shared bit throughput and total time for the KeyGen protocol for $\sigma = 0.707$.

6.2 Secret Keys Generated According to a Centred Binomial Distribution

Finally we examine the case of using the centred binomial distribution for generating the secret keys, with standard deviation selected to be $\sigma = \sqrt{1/2} = 0.707$. This pushes the parameter sizes for the underlying BGV scheme up a little, as we need to cope with more potential noise growth due to the 'heavier' secret key. Using the same analysis as before we find the parameter sizes given in Table 5, with the resulting run times for distributed key generation given in Table 6. This time we performed the experiments also for $n = 4$ and $n = 5$ so as to show how the times grow with n; recall the overall method is $O(n^2)$ as mentioned earlier.

Number of Parties, n	2				3				4				5			
FHE Plaintext Size, $\log_2 p$	64		128		64		128		64		128		64		128	
Polynomial Degree	16384		32768		16384		32768		16384		32768		16384		32768	
$p_0 \vert p_1$ bit length	240	160	369	220	240	160	369	221	242	160	371	221	242	160	371	221

Table 5. Parameter Sizes for secret keys distributed according to a centred binomial distribution, and Gaussian error distribution of $\sigma = 0.707$.

Number of Parties	2		3		4		5	
FHE Plaintext size	64	128	64	128	64	128	64	128
BGV Key Generation	5m30s	19m49s	7m45s	33m00s	12m26s	52m45s	16m34s	2h08m58s

Table 6. Total time for the KeyGen protocol for $\sigma = 0.707$ and secret keys generated according to a centred binomial distribution.

References

1. Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Post-quantum key exchange - A new hope. In: Holz, T., Savage, S. (eds.) USENIX Security 2016. pp. 327–343. USENIX Association (Aug 2016)
2. Aly, A., Cozzo, D., Keller, M., Orsini, E., Rotaru, D., Scholl, P., Smart, N.P., Wood, T.: SCALE-MAMBA v1.5: Documentation (2019), `https://homes.esat.kuleuven.be/~nsmart/SCALE/Documentation.pdf`
3. Aly, A., Orsini, E., Rotaru, D., Smart, N.P., Wood, T.: Zaphod: Efficiently combining LSSS and garbled circuits in SCALE. In: Brenner, M., Lepoint, T., Rohloff, K. (eds.) Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography, WAHC@CCS 2019, London, UK, November 11-15, 2019. pp. 33–44. ACM (2019), `https://doi.org/10.1145/3338469.3358943`
4. Asharov, G., Jain, A., Wichs, D.: Multiparty computation with low communication, computation and interaction via threshold FHE. Cryptology ePrint Archive, Report 2011/613 (2011), `http://eprint.iacr.org/2011/613`
5. Bendlin, R., Damgård, I., Orlandi, C., Zakarias, S.: Semi-homomorphic encryption and multiparty computation. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 169–188. Springer, Heidelberg (May 2011)
6. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient pseudorandom correlation generators: Silent OT extension and more. In: Boldyreva,

A., Micciancio, D. (eds.) CRYPTO 2019, Part III. LNCS, vol. 11694, pp. 489–518. Springer, Heidelberg (Aug 2019)

7. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. In: Goldwasser, S. (ed.) ITCS 2012. pp. 309–325. ACM (Jan 2012)
8. Burra, S.S., Larraia, E., Nielsen, J.B., Nordholt, P.S., Orlandi, C., Orsini, E., Scholl, P., Smart, N.P.: High performance multi-party computation for binary circuits based on oblivious transfer. Cryptology ePrint Archive, Report 2015/472 (2015), `http://eprint.iacr.org/2015/472`
9. Curtis, B.R., Player, R.: On the feasibility and impact of standardising sparse-secret LWE parameter sets for homomorphic encryption. Cryptology ePrint Archive, Report 2019/1148 (2019), `https://eprint.iacr.org/2019/1148`
10. 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. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS 2013. LNCS, vol. 8134, pp. 1–18. Springer, Heidelberg (Sep 2013)
11. 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 (Aug 2012)
12. Data61: MP-SPDZ (2019), `https://github.com/data61/MP-SPDZ`
13. Frederiksen, T.K., Keller, M., Orsini, E., Scholl, P.: A unified approach to MPC with preprocessing using OT. In: Iwata, T., Cheon, J.H. (eds.) ASIACRYPT 2015, Part I. LNCS, vol. 9452, pp. 711–735. Springer, Heidelberg (Nov / Dec 2015)
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 (Apr 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 (Aug 2012)
16. Horowitz, E., Sahni, S.: Computing partitions with applications to the knapsack problem. Journal of the Association for Computing Machinery 21, 277–292 (1974)
17. Ishai, Y., Kilian, J., Nissim, K., Petrank, E.: Extending oblivious transfers efficiently. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 145–161. Springer, Heidelberg (Aug 2003)
18. Keller, M., Orsini, E., Scholl, P.: Actively secure OT extension with optimal overhead. In: Gennaro, R., Robshaw, M.J.B. (eds.) CRYPTO 2015, Part I. LNCS, vol. 9215, pp. 724–741. Springer, Heidelberg (Aug 2015)
19. Keller, M., Orsini, E., Scholl, P.: MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016. pp. 830–842. ACM Press (Oct 2016)
20. Koiliaris, K., Xu, C.: A faster pseudopolynomial time algorithm for subset sum. In: Klein, P.N. (ed.) ACM-SIAM Symposium on Discrete Algorithms, SODA 2017. pp. 1062–1072 (2017)
21. Mouchet, C., Troncoso-Pastoriza, J., Hubaux, J.P.: Computing across trust boundaries using distributed homomorphic cryptography. IACR Cryptology ePrint Archive 2019, 961 (2019), `https://eprint.iacr.org/2019/961`
22. Pan, Y., Zhang, F.: A note on the density of the multiple subset sum problems. Cryptology ePrint Archive, Report 2011/525 (2011), `http://eprint.iacr.org/2011/525`

23. Pisinger, D.: Linear time algorithms for knapsack problems with bounded weights. Journal of Algorithms 33, 1–14 (1999)
24. Rotaru, D., Wood, T.: MArBled circuits: Mixing arithmetic and Boolean circuits with active security. In: Hao, F., Ruj, S., Sen Gupta, S. (eds.) INDOCRYPT 2019. LNCS, vol. 11898, pp. 227–249. Springer, Heidelberg (Dec 2019)
25. Wang, X., Ranellucci, S., Katz, J.: Global-scale secure multiparty computation. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017. pp. 39–56. ACM Press (Oct / Nov 2017)