Adding Distributed Decryption and Key Generation to a Ring-LWE Based CCA Encryption Scheme

Michael Kraitsberg[3], Yehuda Lindell[1,3], Valery Osheter[3], Nigel P. Smart[2,4], and Younes Talibi Alaoui[2]

[1] Bar-Ilan University, Israel,
[2] KU Leuven, Leuven, Belgium.
[3] Unbound Technology, Israel,
[4] University of Bristol, Bristol, UK.
michael.kraitsberg@unboundtech.com, yehuda.lindell@biu.ac.il,
valery.osheter@unboundtech.com, nigel.smart@kuleuven.be,
younes.talibialaoui@kuleuven.be

Abstract. We show how to build distributed key generation and distributed decryption procedures for the LIMA Ring-LWE based post-quantum cryptosystem. Our protocols implement the CCA variants of distributed decryption and are actively secure (with abort) in the case of three parties and honest majority. Our protocols make use of a combination of problem specific MPC protocols, generic garbled circuit based MPC and generic Linear Secret Sharing based MPC. We also, as a by-product, report on the first run-times for the execution of the SHA-3 function in an MPC system.

1 Introduction

Distributed decryption enables a set of parties to decrypt a ciphertext under a shared (i.e. distributed) secret key. Distributed decryption protocols for traditional public key encryption and signature schemes have had a long history of innovation [10–13, 15, 17, 20, 21, 26, 27]. But the research on such protocols for schemes based on Ring-LWE (Learning-With-Errors) has only been started quite recently.

Despite research on Ring-LWE, and Fully/Somewhat Homormorphic Encryption (FHE/SHE) schemes derived from Ring-LWE being relatively new, applications of distributed decryption have found numerous applications already. One of the earliest applications we could find is the two-round passively secure FHE-based multiparty computation (MPC) protocol of Gentry [14]. In this MPC protocol, n parties encrypt their inputs to the MPC computation via an FHE scheme, and broadcast the ciphertexts. All parties can then homomorphically compute the desired function, with the result finally obtained via a distributed decryption. A similar methodology is applied in the multi-key FHE techniques of Asharov et al [4]. These two works only aim for passive security, whereas a similar technique is applied in [22] to obtain low-round MPC via the BMR methodology [5], and in [8] to utilize Gentry's technique via an SHE scheme (and not an FHE scheme) for non-full-threshold access structures (such as the ones considered in this paper). Despite the overal protocols obtaining active security, the distributed decryption procedures in [8,22] are only required to be actively secure up to additive errors. A similar situation occurs in the SPDZ protocol [9] in which an actively secure (up to additive errors) distributed decryption protocol is required to produce the multiplication triples in the offline phase. The same technique is used in the High Gear variant of the Overdrive [18] offline phase for SPDZ.

The application of such distributed decryption protocols is however not just restricted to usage in MPC protocols. It is well established in the side-channel community that such key-splitting techniques can form the basis of various defences based on 'masking'; see for example a recent initiative of NIST in this area [7]. However, every masking technique requires a recombination technique, which is exactly what distributed decryption provides.

More importantly, the interest in standard Ring-LWE schemes due to the need to find suitable Post-Quantum Crypto (PQC) algorithms, as evidenced by the current NIST "competition", means that PQC schemes which can support distributed decryption will become of more interest. However there is a major issue with prior techniques to this problem. Firstly the methods require "noise-flooding" to ensure that no information leaks about the underlying secret key during decryption. This requires that the ciphertext modulus q needs to be made much larger than in a standard system which does not support distributed decryption. Secondly, the methods are only actively secure up

to additive error (i.e. they are not fully actively secure) and they only allow distributed decryption of the IND-CPA versions of the underlying encryption schemes.

In this paper we present efficient methods to perform actively secure distributed decryption for IND-CCA versions of Ring-LWE based encryption in the case of three party honest majority protocols. This is done by combining in a novel manner traditional garbled circuit based MPC, with bespoke protocols for the specific functionality we require. On the way we also provide, to our knowledge, the first MPC implementation of the evaluation of the SHA-3 function; previously in [29] an MPC-optimized circuit was given, but no execution times for an actual MPC evaluation of SHA-3 has been presented. We also show how to utilize secret sharing based protocols such as embodied in the SCALE-MAMBA system [3] to produce an efficient distributed key generation procedure suitable for our underlying distributed decryption procedure.

Prior work on CCA secure distributed decryption protocols, even for encryption schemes based on "traditional" assumptions (such as RSA or Discrete Logarithms), such as those in [12, 20, 27], have looked at designing special purposes encryption procedures which are both CCA secure, and which enable an efficient distributed decryption procedure. In this work we instead take an off-the-shelf CCA secure encryption scheme and show how it can be made into a scheme which supports both distributed decryption and distributed key generation. This brings added challenges as the method for ciphertext checking is not immediately "MPC-friendly". Despite this drawback we show that such schemes can be implented via MPC.

In total we use *four* different types of secret sharing between the three parties in order to obtain efficient protocols for the various sub-steps:

1. INS-sharing [16] modulo q of the Ring-LWE secret key.
2. Shamir secret sharing modulo q to generate the INS secret key via SCALE-MAMBA.
3. An additive 3-party binary sharing of the output of the Round function, before we pass it to the KMAC operation. This additive sharing is non-standard and looks like a cross between INS and replicated sharing.
4. An additive sharing modulo q of the output of the KMAC operation between two of the parties S_1 and S_2.

To illustrate our methods in terms of a concrete Ring-LWE system we take the LIMA submission [1] to the PQC contest. We take the latest version of this submission (version 1.1 at the time of writing). However, almost all our techniques will apply, with minor changes to the specific definitions of usage of SHA-3 etc, to a number of the other Ring-LWE systems under submission. A major advantage of the LIMA proposal versus a proposal such as say NTRU is that key generation and encryption are essentially linear operations; thus providing a distributed actively secure protocol for key generation and re-encryption becomes easy. For NTRU the key generation method, for example, needs to generate polynomials with distributions which cannot be generated in a linear fashion; in particular distributions with given coefficients with a given weight of -1 and $+1$ coefficients.

We end this section by noting that running our distributed decryption protocol using two as opposed to three parties, and using traditional passively secure Yao protocols, results in a passively secure distributed decryption protocol. For the two party case of the key generation protocol we could utilize the SPDZ [9] implementation within the SCALE-MAMBA framework. The two-party key generation would be actively secure and, more importantly, is possible since the modulus $q = 40961$ used in the LIMA v1.1 scheme is "FHE-Friendly" and hence can be used as the plaintext modulus for the SPDZ-offline phase.

1.1 Overview of Prior Techniques

To understand prior techniques for distributed decryption in the Ring-LWE setting, we will consider a simple BGV style Ring-LWE [6] ciphertext, under a secret key which is distributed with respect to a full threshold access structure. However, it will be easily seen that this prior work extends to any distribution under a linear secret sharing scheme. The BGV Ring-LWE encryption scheme is parametrized by a global ring R (usually a cyclotomic ring of integers of degree N over $\mathbb{Z}$), a plaintext modulus p, a ciphertext modulus q, and distributions χ_i (which produce "small" elements in the ring R). We let R_q denotes the ring R when considered modulo q. A secret key is an element $s \in R$, and a public key is given by a pair $\mathfrak{pk} = (a, b) \in R^2$ where a is chosen at random from R and $b = a \cdot s + e \pmod q$ where $b \leftarrow \chi_0$.

To encrypt a message $m \in R_p$ the encryptor selects random values $e_i \in \chi_i$ and computes

$$c_0 = b \cdot v + p \cdot e_1 + m \pmod{q} \quad \text{and} \quad c_1 = a \cdot v + p \cdot e_2.$$

A ciphertext is then given by the pair $\mathfrak{c} = (c_0, c_1)$. If one considers a (somewhat) homomorphic variant of the above then all valid ciphertexts (i.e. ones which will always validly decrypt) will be of the form

$$c_0 = c_1 \cdot s + p \cdot e + m \pmod{q}$$

where e comes from some other distribution χ (which depends on the function which has been evaluated). It is readily seen that processed, and fresh, ciphertexts can be decrypted by computing

$$m = (c_0 - s \cdot c_1 \pmod{q}) \pmod{p}.$$

Decryption will be valid as long as the $\|c_0 - s \cdot c_1 \pmod{q}\|_\infty \leq B$, and a value of q just larger than $2 \cdot B$ is chosen so that this is guaranteed to produce no wrap-around.

Now consider the situation where the secret key s has been additively shared amongst n parties, i.e. P_1 holds s_1, P_2 holds s_2 and so on, such that $s = s_1 + \cdots + s_n \pmod{n}$. A naive method to perform such a distributed decryption would be for each party P_i to compute a 'decryption share' d_i as

$$d_i = \begin{cases} c_0 - s_1 \cdot c_1 + p \cdot e_1' \pmod{q} & \text{If } i = 1 \\ -s_i \cdot c_1 + p \cdot e_i' \pmod{q} & \text{If } i \neq 1 \end{cases}$$

where e_i' comes from some LWE distribution χ_i'. On broadcasting these values all parties can compute

$$m = \left(\sum_{i=1}^{n} d_i \pmod{q} \right) \pmod{p}.$$

Naively this seems to be (passively) secure, as the values d_i seem to be indistinguishable from uniform under the LWE assumption, and thus hide s_i, however the partial sum

$$\sum_{i=1}^{n} d_i \pmod{q}$$

reveals information about s. To see this, consider the decryption equation for a fresh BGV ciphertext.

This naive method does however form the basis of prior techniques to perform this operation; see [4, 9] for the full threshold case and [8] for the case of threshold access structures. To remove the information leakage on s from the sum, the prior work applies the Smuddging Lemma from [4]. In essence, one selects a statistical security parameter secp, and we now choose q so that $q > 2 \cdot (1 + n \cdot 2^{\mathsf{secp}}) \cdot B$. To produce the partial decryptions each party then computes decryption shares d_i as before, except now the e_i' are selected uniformly from ring elements with infinity norm bounded by $2^{\mathsf{secp}} \cdot B/p$. Then, when we form the partial sum $\sum_{i=1}^{n} d_i \pmod{q}$ we have added in an essentially uniform random element which masks any information about s to the parties.

It should be noted that the statistical security in this step is really only per coefficient of the ring-element, and thus to obtain "true" security one needs to actually choose secp much larger than is done in (say) the SCALE-MAMBA system [3] (where a value of $\mathsf{secp} = 40$ is selected). Although in practice this is probably not a problem as it is unknown how to extract the minor statistical variations per coefficient when processing a number of full ring elements in this manner. Even when selecting low values of secp such as 40, one obtains a huge blow-up in the parameters of the scheme and hence increased computational cost.

In addition this method cannot be applied to distributed decryption of encryptions under an IND-CCA encryption scheme. For this one needs to only release the message once the "ciphertext checking" has passed, and thus the checking of the ciphertext for validity needs to be performed within the distributed decryption procedure using some form of MPC protocol. It is to this latter problem we address this work, and show how we can achieve this in the case of a three party honest majority protocol; without the need to increase the parameter q.

2 Preliminaries

To focus our discussion we pick a specific Ring-LWE submission to the NIST PQC "competition"; in particular v1.1 of LIMA [1]. This was selected as it utilizes a relatively standard transform for ciphertext validity checking; namely randomness recovery followed by re-encryption which could pose a particular problem for a distributed decryption protocol. In addition the encryption and key generation procedures are also relatively linear, allowing one to utilize simple MPC protocols to perform re-encryption and distributed key generation.

We will focus on the main parameter sets in the LIMA proposal. In particular $N = 1024$ and $q = 40961$. However, our protocol can be easily adapted to the other parameter sets (in fact the protocol stays the same, all that needs to change is the specific implementation of the underlying garbled and arithmetic circuits).

In this section we summarize the LIMA construction and also summarize some tools from the MPC literature which we will be using. Our focus in the MPC literature will be on three party honest majority protocols which offer active security *with abort*.

2.1 The LIMA IND-CCA Encryption Scheme

Here we summarize the LIMA v1.1 construction. For more details readers are referred to [1]. As explained above we use the latest version of the proposal which avoids the rejection sampling in encryption of the first proposal and has, as a result, smaller parameters.

Cycloctomic Rings: LIMA makes use of two types of cyclotomic rings, in this paper we will concentrate on only the first type (minor modifications in what follows are needed to support the second type proposed in LIMA). We select N to be power of two, q to be a prime such that $q \equiv 1 \pmod{2 \cdot N}$. The scheme makes use of the following rings

$$R = \mathbb{Z}[X]/(X^N + 1), \quad R_2 = \mathbb{Z}_2[X]/(X^N + 1), \text{ and } R_q = \mathbb{Z}_q[X]/(X^N + 1).$$

Note that $\Phi_{2 \cdot N}(X) = X^N + 1$ in this case. Elements of these rings are degree $(N-1)$ polynomials with coefficients from $\mathbb{Z}, \mathbb{Z}_2, \mathbb{Z}_q$, respectively. Equivalently, these are represented as vectors of length N, with elements in the $\mathbb{Z}, \mathbb{Z}_2, \mathbb{Z}_q$, respectively.

LIMA makes a lot of use of the number theoretic FFT algorithm to enable fast multiplication of ring elements. We will denote this operation in this paper by $\mathbf{f} \leftarrow \mathsf{FFT}(f)$ for the forward FFT, and $f \leftarrow \mathsf{FFT}^{-1}(\mathbf{f})$ for the inverse FFT operation. The forward direction maps a polynomial of degree $N-1$ into a vector of length N over the same finite field $\mathbb{F}_q$ (by choice of q). For our MPC operations it is important to note that the FFT operation is a linear operation, i.e. $\mathsf{FFT}(f+g) = \mathsf{FFT}(f) + \mathsf{FFT}(g)$.

Use of SHA-3 in KMAC256: LIMA makes use of KMAC256, to create an XOF (Extendable Output Function) and a KDF (Key Derivation Function). The algorithm KMAC256 is itself derived from the SHA-3 hash function and is defined in NIST SP 800 185 [25]. Following the LIMA specification we use the following notation for the various uses of KMAC256. When called in the form $\mathsf{XOF} \leftarrow \mathsf{KMAC}(\mathsf{key}, \mathsf{data}, 0)$ the output is an XOF object, and when called in the form $K \leftarrow \mathsf{KMAC}(\mathsf{key}, \mathsf{data}, L)$ the output is a string of L bits in length. In both cases the input is a key key (of length at least 256 bits), a (one-byte) data string data, and a length field L in bits. The data string data is a diversifier and corresponds to the *domain separation* field in the KMAC standard. Different values of data will specify different uses of the KMAC construction. In the case when $L = 0$ we shall let $a \leftarrow \mathsf{XOF}[n]$ denote the process of obtaining n bytes from the XOF object returned by the call to KMAC. The KDF in LIMA is given by the notation $\mathsf{KDF}^{[n]}(k)$, which outputs the result of computing $\mathsf{KMAC}(k, 0x00, n)$.

In Figure 1 we describe how LIMA uses the XOF to generate random values in different domains and with different distributions. These are $a \underset{\mathsf{XOF}}{\leftarrow} \mathbb{F}_q$ to generate uniformly random single finite field element, $\mathbf{a} \underset{\mathsf{XOF}}{\leftarrow} \mathbb{F}_q^n$ to generate a vector of such elements, and $\mathsf{GenerateGaussianNoise}_{\mathsf{XOF}}(\sigma)$ to generate elements in $\mathbb{F}_q$ from a distribution which is an approximation to a discrete Gaussian. It will turn out their method here is particularly well suited to enabling distributed key generation. In particular the LIMA algorithm uses the method of approximating a Discrete Gaussian via a centred binomial distribution given in [2], to produce a Gaussian with standard deviation about 3.19.

$a \xleftarrow[\mathsf{XOF}]{} \mathbb{F}_q$

1. $s \leftarrow \mathsf{XOF}[\, 2 \cdot \lceil \log_{256} q \rceil \,]$.
2. Convert s to an integer (msb is the left most bit).
3. $a \leftarrow s \pmod q$.
4. Output a.

$\mathbf{a} \xleftarrow[\mathsf{XOF}]{} \mathbb{F}_q^n$

1. For i from 1 to n do
   (a) $a_i \xleftarrow[\mathsf{XOF}]{} \mathbb{F}_q$.
2. Output $\mathbf{a}$.

$\mathsf{GenerateGaussianNoise}_{\mathsf{XOF}}(\sigma)$

1. $t \leftarrow \mathsf{XOF}[5]$; interpretting t as a bit string of length 40.
2. $s \leftarrow 0$.
3. For $i = 0$ to B do
   (a) $s \leftarrow s - t[2 \cdot i] + t[2 \cdot i + 1]$.
4. Return s.

Figure 1. Using of a XOF to generate random values

$\mathsf{KeyGen}(N)$

1. $a = (a_0, \ldots, a_{N-1}) \xleftarrow[\mathsf{XOF}]{} \mathbb{F}_q^N$.
2. For $i = 0$ to $N - 1$ do $s_i \leftarrow \mathsf{GenerateGaussianNoise}_{\mathsf{XOF}}(\sigma)$.
3. For $i = 0$ to $N - 1$ do $e'_i \leftarrow \mathsf{GenerateGaussianNoise}_{\mathsf{XOF}}(\sigma)$.
4. $\mathbf{a} \leftarrow \mathsf{FFT}(a), \mathbf{s} \leftarrow \mathsf{FFT}(s), \mathbf{e}' \leftarrow \mathsf{FFT}(e')$.
5. $\mathbf{b} \leftarrow (\mathbf{a} \otimes \mathbf{s}) \oplus \mathbf{e}'$,
6. $\mathfrak{sk} \leftarrow (\mathbf{s}, \mathbf{a}, \mathbf{b})$.
7. $\mathfrak{pk} \leftarrow (\mathbf{a}, \mathbf{b})$.
8. Return $(\mathfrak{pk}, \mathfrak{sk})$

Figure 2. LIMA Key Generation

LIMA Key Generation: The specification of LIMA details that the private component of a public/private key pair are generated from the KMAC256 XOF. However, in practice this component can come from any source of random bits, thus the XOF output in lines 2 and 3 of Figure 2 can be replaced by *any* source of random bits known to the secret key creator. We will make use of this fact in our distributed key generation procedure later on. Key Generation proceeds as in Figure 2, where we assume a XOF has already been initiated and the operations $\otimes$ and $\oplus$ denote pointwise multiplication and addition $\pmod q$.

Enc-CPA-Sub$(\mathbf{m}, \mathfrak{pk}, \mathsf{XOF})$:

1. $\ell = |\mathbf{m}|$.
2. If $\ell > N$ then return $\perp$.
3. $\mu \leftarrow \mathsf{BV\text{-}2\text{-}RE}(\mathbf{m})$,
4. For $i = 0$ to $N-1$ do $v_i \leftarrow \mathsf{GenerateGaussianNoise}_{\mathsf{XOF}}(\sigma)$.
5. For $i = 0$ to $N-1$ do $e_i \leftarrow \mathsf{GenerateGaussianNoise}_{\mathsf{XOF}}(\sigma)$.
6. For $i = 0$ to $N-1$ do $d_i \leftarrow \mathsf{GenerateGaussianNoise}_{\mathsf{XOF}}(\sigma)$.
7. $\mathbf{v} \leftarrow \mathsf{FFT}(v)$, $\mathbf{e} \leftarrow \mathsf{FFT}(e)$.
8. $x \leftarrow d + \Delta_q \cdot \mu \pmod q$.
9. $s \leftarrow \mathsf{FFT}^{-1}(\mathbf{b} \otimes \mathbf{v})$.
10. $t \leftarrow s + x$.
11. $c_0 \leftarrow \mathsf{Trunc}(t, \ell)$.
12. $\mathbf{c}_1 \leftarrow (\mathbf{a} \otimes \mathbf{v}) \oplus \mathbf{e}$.
13. Output $\mathbf{c} = (c_0, \mathbf{c}_1)$.

Dec-CPA$(\mathbf{c}, \mathfrak{sk})$:

1. Define ℓ to be the length of c_0.
2. If $\ell \neq 0 \pmod 8$ then return $\perp$.
3. $v \leftarrow \mathsf{FFT}^{-1}(\mathbf{s} \otimes \mathbf{c}_1)$.
4. $t \leftarrow \mathsf{Trunc}(v, \ell)$.
5. $f \leftarrow c_0 - t$.
6. Convert f into centered-representation modulo q.
7. $\mu \leftarrow \left| \left\lfloor \frac{2}{q} f \right\rceil \right|$
8. $\mathbf{m} \leftarrow \mathsf{RE\text{-}2\text{-}BV}(\mu)$.
9. Return $\mathbf{m}$.

Figure 3. Base LIMA Encryption and Decryption Operations

Encryption and Decryption: Both the CCA encryption and decryption operations make use of a sub-procedure, called Enc-CPA-Sub$(\mathbf{m}, \mathfrak{pk}, \mathsf{XOF})$ which takes as input a message in $\mathbf{m} \in \{0,1\}^\ell$, a public key and an initialized XOF object XOF, and outputs a ciphertext $\mathfrak{c}$. The associated inverse operation is denoted Dec-CPA$(\mathbf{c}, \mathfrak{sk})$. These algorithms are defined in Figure 3. These operations make use of three sub-routines:

- Trunc denotes a procedure which throws away unnecessary coefficients of c_0, retaining only the ℓ elements corresponding to message component.
- BV-2-RE is a procedure which takes a bit string of length at most N and maps it into R_2.
- RE-2-BV is the inverse operation to BV-2-RE.

We can then define the CCA LIMA encryption, decryption, encapsulation and decapsulation operations as in Figure 4

2.2 Three Party Honest Majority MPC Using Garbled Circuits

Our protocols make use of actively secure garbled circuit based MPC for honest majority in the three party setting. In this situation we use the techniques from [24]. The basic protocol to evaluate a function F on inputs x_1, x_2 and x_3

Enc-CCA($\mathbf{m}, \mathfrak{pk}, \mathbf{r}$):

1. If $|\mathbf{r}| \neq 256$ or $|\mathbf{m}| \geq N - 256$ then return $\perp$.
2. $\mu \leftarrow \mathbf{m} \| \mathbf{r}$.
3. XOF $\leftarrow$ KMAC($\mu, 0x03, 0$).
4. $\mathbf{c} \leftarrow$ Enc-CPA-Sub($\mu, \mathfrak{pk}$, XOF).
5. Return $\mathbf{c}$.

Dec-CCA($\mathbf{c}, \mathfrak{sk}$):

1. $\mu \leftarrow$ Dec-CPA($\mathbf{c}, \mathfrak{sk}$).
2. If $|\mu| < 256$ then return $\perp$.
3. XOF $\leftarrow$ KMAC($\mu, 0x03, 0$).
4. $\mathbf{c}' \leftarrow$ Enc-CPA-Sub($\mu, \mathfrak{pk}$, XOF).
5. If $\mathbf{c} \neq \mathbf{c}'$ then return $\perp$.
6. $\mathbf{m} \| \mathbf{r} \leftarrow \mu$, where $\mathbf{r}$ is 256 bits long.
7. Return $\mathbf{m}$.

Encap-CCA($\ell, \mathfrak{pk}, \mathbf{s}$):

1. If $|\mathbf{r}| < 384$ or $|\mathbf{r}| > N$ then return $\perp$.
2. XOF $\leftarrow$ KMAC($\mathbf{r}, 0x05, 0$).
3. $\mathbf{c} \leftarrow$ Enc-CPA-Sub($\mathbf{r}, \mathfrak{pk}$, XOF).
4. $\mathbf{k} \leftarrow \mathsf{KDF}^{[\ell]}(\mathbf{r})$.
5. Return $(\mathbf{c} = (c_0, \mathbf{c}_1), \mathbf{k})$.

Decap-CCA($\ell, \mathbf{c}, \mathfrak{sk}$):

1. $\mathbf{r} \leftarrow$ Dec-CPA($\mathbf{c}, \mathfrak{sk}$).
2. If $|\mathbf{r}| < 384$ then return $\perp$.
3. XOF $\leftarrow$ KMAC($\mathbf{r}, 0x05, 0$).
4. $\mathbf{c}' \leftarrow$ Enc-CPA-Sub($\mathbf{r}, \mathfrak{pk}$, XOF).
5. If $\mathbf{c} \neq \mathbf{c}'$ then return $\perp$.
6. $\mathbf{k} \leftarrow \mathsf{KDF}^{[\ell]}(\mathbf{r})$.

Figure 4. CCA Secure Encryption, Decryption, Encapsulation and Decapsulation Algorithms for LIMA

from parties P_1, P_2 and P_3 is as follows. Parties P_1 and P_2 agree on a random seed s and then use s to generate a garbled circuit. Party P_3 acts as evaluator. If P_1 or P_2 cheats then this is detected by P_3 as they will obtain different circuits, where as if P_3 cheats in sending output tables values incorrectly back to P_1 or P_2 (for their output), then the table values will not decode correctly. The overall protocol is described in Figure 5. Thus we cheaply obtain active security with abort in this scenario.

1. Denote the inputs of S_1, S_2 and S_3 by x_1, x_2 and x_3, respectively.
2. Let $f(x_1, x_2, x_3)$ be the function to be computed and let $C(x_1, x_2, x_3)$ be a circuit that computes f.
3. Party S_1 chooses a random seed s and generates a garbled circuit GC computing C and the output translation tables O_1, O_2 and O_3 for the three parties outputs), using seed s.
4. Party S_3 chooses a random $x_3^a \leftarrow \{0,1\}^{|x_3|}$ and sets $x_3^b \leftarrow x_3^a \oplus x_3$ and it sends x_3^a to S_1, and sends x_3^b to S_2.
5. Party S_1 sends (GC, O_3) to S_3. For every wire w associated with the input x_3 in the circuit, the garbled circuit definition is assumed to includes $(H(k_w^0), H(k_w^1))$ in this order (where the keys on that wire are k_w^0, k_w^1).
6. Party S_1 sends s to S_2.
7. Party S_2 computes GC from s and sends $h = H(GC)$ to S_3.
8. S_3 checks that $h = H(GC)$ where h is the value it received from S_2, and GC is as received from S_1. If not, it aborts.
9. Party S_1 sends S_3 the keys associated with its own input x_1 and with x_3^a.
10. Party S_2 sends S_3 the keys associated with its own input x_2 and with x_3^b.
11. Parties S_1 and S_2 run a secure tossing protocol to generate random strings $R_1, \ldots, R_{|x_3|}$.
12. Denote the bits of x_3^a by $x_3^a[1], \ldots, x_3^a[\ell]$, denote the bits of x_3^b by $x_3^b[1], \ldots, x_3^b[\ell]$, and denote the wires associated with x_3 by $w_1, \ldots, w_\ell$.
13. For every $i = 1, \ldots, \ell$, party S_1 sends $k_i' \leftarrow k_{w_i}^0 \oplus x_3^a[i] \cdot \Delta \oplus R_i$ to S_3. We use the "free-XOR" trick so we have $k_{w_i}^1 = k_{w_i}^0 \oplus \Delta$.
14. For every $i = 1, \ldots, \ell$, party S_2 sends $k_i'' \leftarrow x_3^b[i] \cdot \Delta \oplus R_i$ to S_3.
15. For every $i = 1, \ldots, \ell$, party S_3 computes $k_{w_i}^{x_3[i]} \leftarrow k_i' \oplus k_i''$ and checks that this is consistent with the appropriate hash. That is, if $x_3[i] = 0$ then the hash of the key equals the first value in the appropriate pair; otherwise the second value. If not then S_3 aborts.
16. S_3 computes the garbled circuit GC and obtains their output using the table O_3.
17. S_3 sends the garbled outputs corresponding to S_1 and S_2's output wires to S_1 and S_2 respectively.
18. S_1 and S_2 decode their output using the translation tables O_1 and O_2.

Figure 5. Garbled Circuit Based Three Party Computation

2.3 Three Party Honest Majority MPC Using Shamir Secret Sharing

We also require honest majority three party actively secure MPC with abort based on linear secret sharing over the finite field $\mathbb{F}_q$. For this we use a protocol based on Shamir secret sharing implemented in the SCALE-MAMBA system [3]. This uses an offline phase to produce multiplication triples and shared random bits (using Maurer's multiplication protocol [23]) and then an online phase which checks for correctness by using the error detection properties of the underlying Reed-Solomon codes. See [19, 28] for precise details of how this protocol works. This arithmetic circuit based MPC protocol is used in our distributed key generation phase, and we make crucial use of the ability of the SCALE-MAMBA system to generate shared random bits in the offline phase; as then our online phase becomes essentially a local operation.

We denote secret shared values over $\mathbb{F}_q$ by the notation $[a]$. In this notation linear operations, such as $[z] \leftarrow \alpha \cdot [x] + \beta \cdot [y] + \gamma$ are local operations, and hence essentially for free. Where as non-linear operations, such as $[z] \leftarrow [x] \cdot [y]$ require interaction. In the SCALE-MAMBA system these are done using a pre-processed set of Beaver triples $([a], [b], [c])$ with $c = a \cdot b$.

Output/opening values to all players will be denoted by $\mathsf{Output}([a])$ by which we mean all players learn the value of a, and abort if the value is not output correctly. Outputing to a specific player we will denote by $\mathsf{Output\text{-}To}(i, [a])$, in which case player i will learn the value of a, and abort if the value is not correct.

One can also use these pre-processed Beaver triples to generate random shared elements (by taking a triple and using $[a]$ and $[b]$ as the random elements. Of course when using Shamir sharing one could also generate such sharings

using a PRSS, however the SCALE-MAMBA system does not currently support this functionality. So when generating random elements in the online phase we simply consume the first two components of a Beaver triple, and we will write $([a], [b], [c]) \leftarrow \mathsf{Triples}$, this is secure as long as $[c]$ is never used later.

The offline phase also produced shared random bits, namely sharings of the form $[b]$ with $b \in \{0, 1\}$. We will denote this operation in what follows as $[b] \leftarrow \mathsf{Bits}$.

3 SHA3 in MPC

The TinyGarble compiler [29] has been reported to produce a circuit for the SHA-3 core internal Keccak-f function of 38,400 AND gates (160,054 total gates). Using similar techniques to the TinyGarble paper we compiled our own circuit for Keccak-f, finding a circuit with the same number of gates and 193,686 wires. This function takes as input, a sequence of 1600 bit values, and returns a 1600 bit value. The output is either then passed into the next round, during the absorption phase where it is combined with additional input, or part of the output is used as the output of SHA-3, in the squeezing phase.

Using our garbled circuit based protocol for honest majority computation amongst three parties, we were able to execute the Keccak-f function with a latency of 16ms per operation. With the testing being conducted on a set of three Linux RHEL servers running on AWS of type `t2.small`, which correspond to one "virtual CPU" and 2GB of RAM.

4 Distributed Decryption for CCA-Secure Ring-LWE Encryption

Recall that a public key is a pair $(\mathbf{a}, \mathbf{b})$ and a secret key is a value $\mathbf{s}$, where $\mathbf{a}, \mathbf{b}, \mathbf{s} \in \mathbb{Z}_q^N$. Given our three servers, of which we assume at least two are honest, we share the secret key using Ito–Nishizeki–Saito sharing [16]. In particular S_1 is assumed to hold $(\mathbf{s}_1^{1,2}, \mathbf{s}_1^{1,3}) \in \mathbb{Z}_q^N$, S_2 is assumed to hold $(\mathbf{s}_2^{1,2}, \mathbf{s}_1^{2,3}) \in \mathbb{Z}_q^N$, and S_2 is assumed to hold $(\mathbf{s}_2^{1,3}, \mathbf{s}_2^{2,3}) \in \mathbb{Z}_q^N$ such that

$$\mathbf{s}_1^{1,2} + \mathbf{s}_2^{1,2} = \mathbf{s}_1^{1,3} + \mathbf{s}_2^{1,3} = \mathbf{s}_1^{2,3} + \mathbf{s}_2^{2,3} = \mathbf{s}.$$

How one generates a valid secret key satisfying this secret sharing we discuss in the next section. We call such a sharing an INS-sharing of $\mathbf{s}$. Our overall distributed decryption and decapsulation protocols are then build out of a number of special protocols which either utilize our generic 3-party garbled circuit based protocol from earlier, or utilize special purpose MPC protocols built on top the ISN-sharing of inputs or other sharings of inputs. Thus in this protocol we combine a variety of MPC techniques together in a novel manner.

4.1 Sub-Protocol: Round Function

We first require a protocol which takes an ISN-sharing of a vector $\mathbf{f}$ and produces the output of the function

$$\mu \leftarrow \left| \left\lfloor \frac{2}{q} f \right\rceil \right|$$

from the procedure $\mathsf{Dec\text{-}CPA}(\mathbf{c}, \mathfrak{sk})$. In particular it needs to evaluate the functionality given in Figure 6, which we do via the protocol given in Figure 7

We note that this protocol is secure by definition since the only thing defined here is the circuit, and a protocol that is secure for malicious adversaries is used to compute it. Let $|q|$ denotes the number of bits needed to represent q and recall that addition and each less-than-comparison can be computed using a single AND gate per bit. Thus, $a + b \bmod q$ can be computed using exactly $4 \cdot |q|$ AND gates, and all the initial additions require $12 \cdot |q|$ AND gates. Next, the bitwise NOR of v, w requires $2 \cdot |q| - 1$ AND gates, each of the 2 less-than-comparisons (and greater-than etc.) of c are computed using $|q|$ AND gates, and there is 1 more AND gate. Overall, we have therefore have a cost of $12 \cdot |q| + 2 \cdot |q| - 1 + 2 \cdot |q| + 1 = 16 \cdot |q|$ AND gates. In our experiments we used the parameter set of LIMA with $q = 40961$ and thus $|q| = 16$. Hence, each execution of this protocol for an individual coefficient requires 256 AND gates. When iterated over the 1024 coefficients we end up with a total of $262,144$ AND gates.

Input:

1. S_1 has $f_1^{1,2}, f_1^{1,3} \in \mathbb{Z}_q^\ell$.
2. S_2 has $f_2^{1,2}, f_1^{2,3} \in \mathbb{Z}_q^\ell$.
3. S_3 has $f_2^{1,3}, f_2^{2,3} \in \mathbb{Z}_q^\ell$.

Computation:

1. Compute $f^{1,2} = f_1^{1,2} + f_2^{1,2} \bmod q$.
2. Compute $f^{1,3} = f_1^{1,3} + f_2^{1,3} \bmod q$.
3. Compute $f^{2,3} = f_1^{2,3} + f_2^{2,3} \bmod q$.
4. Compute $b = 1$ if $f^{1,2} = f^{1,3} = f^{2,3}$ and $b = 0$ otherwise.
5. Convert f into centered representation.
6. Compute $\mu \leftarrow \left| \left\lfloor \frac{2}{q} f \right\rceil \right|$.
7. Choose random $\mu_1, \nu_1 \leftarrow \{0,1\}^\ell$ and set $\mu_2 \leftarrow \mu_1 \oplus \mu$ and $\nu_2 \leftarrow \nu_1 \oplus \mu$ (where here we mean bitwise XOR).

Output:

1. All parties S_1, S_2, S_3 receive b.
2. If $b = 1$ then S_1 receives μ_1, ν_1, S_2 receives μ_2, ν_2, and S_3 receives μ_2, ν_1.

Figure 6. The Functionality : $\mathcal{F}_{\mathsf{Round}}$

4.2 Sub-Protocol: Secure Evaluation of the Enc-CPA-Sub Function

Our next sub-protocol is to evaluate the Enc-CPA-Sub function on inputs which have been INS-shared. The protocol is given in Figure 9 but from a high level works as follows: Firstly the three parties execute the KMAC function on their suitably padded inputs (which have been shared via a different secret sharing scheme), from this S_1 and S_2 obtain an additive $\mathbb{F}_q$-sharing of the output bits. This operation utilizes the SHA-3 implementation given earlier as a sub-procedure, and to aid readability we separate this operation into a sub-protocol in Figure 8. In this protocol the parties have as input a sharing of a bit string μ defined as follows: S_1 holds (μ_1, ν_1), S_2 holds (μ_1, ν_2), and S_3 holds (μ_2, ν_1) such that $\mu = \mu_1 \oplus \mu_2 = \nu_1 \oplus \nu_2$. The output of the function will be an $\mathbb{F}_q$-sharing between S_1 and S_2 of the XOF applied to this input with diversifier D. The diversifier will be $0x03$ for decryption and $0x05$ for decapsulation). Note, that the first thing the circuit does is to ensure the input values are consistent, i.e. $\mu_1 \oplus \mu_2 = \nu_1 \oplus \nu_2$. Also note that only party S_2 obtains output from this step. Since the number of AND gates in the permutation function is 38,400 and we have 114 rounds, then the total number of AND gates needed to execute this step is *approximately* $114 \cdot 38,400 = 4,377,600$, plus the number of AND gates needed to create S_2's output (which is approximately $3 \cdot 40 \cdot N \cdot \log_2 q \approx 1,966,080$).

The additive $\mathbb{F}_q$-sharing between S_1 and S_2 output from Figure 8 is then used in a completely local manner by S_1 and S_2 to obtain a modulo q additive sharing of the supposed ciphertext. The fact we can perform mainly local operations is because the method to generate approximate Gaussian noise is completely linear and the FFT algorithm is itself linear. This is then revealed to players S_1 and S_2, via means of a garbled circuit computation between the three players. See Figure 9 for details.

The privacy of the protocol to evaluate Enc-CPA-Sub is inherent in the fact we use secure actively secure protocols to evaluate the two required garbled circuits. The only place that an active adversary could therefore deviate is by entering incorrect values into the evaluation of the Trunc function; e.g. S_1 could enter the incorrect value for μ_1 or $y^{(1)}$. Any incorrect adversarial behaviour here will either result in an incorrect value of c_0', which will be detected by the calling algorithm.

4.3 Secure Evaluation of Dec-CCA$(\mathbf{c}, \mathfrak{sk})$

We can now give the method for decryption for the CCA public key encryption algorithm, see Figure 10. The secret key $\mathfrak{sk}$ is shared as described earlier, with the ciphertext $\mathbf{c} = (c_0, \mathbf{c}_1)$ being public.

We first, define a boolean circuit C that receives for input bit α, β and six elements $a, b, c, d, e, f \in \mathbb{Z}_q$, and works as follows:

1. Compute $x = a + b \bmod q$, $y = c + d \bmod q$ and $z = e + f \bmod q$. Observe that addition modulo q inside a circuit works by first adding two numbers (into a value that is larger by one bit) and then subtracting q if the comparison of the result with q is returns a value one. In a circuit this looks like:

$$a + b - (LT(a + b, q) \wedge q)$$

   where $LT(r, s) = 0$ if and only if $r < s$ (as integers). Note that we compute the AND of a single bit $LT(a + b, q)$ and a value q; our meaning is to compute the AND of each bit of q.
2. Compute $v = x \oplus y$ and $w = x \oplus z$. Denote by γ the bitwise NOR of all the bits of v and w (i.e., $\gamma = 1$ if and only if *all* the bits of v and w equal 0, meaning that $x = y$ and $x = z$ and so $x = y = z$).
3. Let δ be the bit that satisfies the following Boolean formula:

$$\delta = \left[\left(x \geq \frac{q-1}{4}\right) \wedge \left(x < \frac{3(q-1)}{4}\right)\right]$$

4. Output β on the first output wire.
5. Output γ on the second output wire.
6. Output $\gamma \wedge (\alpha \oplus \delta)$ on the third output wire.
7. Output $\gamma \wedge (\beta \oplus \delta)$ on the fourth output wire.

Now, the function Round can be computed securely, as follows:

1. S_1 chooses random $\mu_1, \nu_1 \leftarrow \{0,1\}^\ell$; denote the ith bit of μ_1 by μ_1^i and the ith bit of ν_1 by ν_1^i.
2. For every $i = 1, \ldots, \ell$:
   (a) Denote by $f[i]$ the ith element of $f \in \mathbb{Z}_q^\ell$.
   (b) S_1, S_2 and S_3 securely compute $C\left(\mu_1^i, \nu_1^i, f_1^{1,2}, f_2^{1,2}, f_1^{1,3}, f_2^{1,3}, f_1^{2,3}, f_2^{2,3}\right)$ using the previous 3-party garbled circuit protocol. Note that S_1 provides input $(\mu_1^i, \nu_1^i, f_1^{1,2}, f_1^{1,3})$, S_2 provides input $(f_2^{1,2}, f_1^{2,3})$, and S_3 provides input $(f_2^{1,3}, f_2^{2,3})$. The secure protocol provides outputs as follows:
      i. S_3 *only* receives the output bit β on the first output wire. S_3 denotes this output by ν_1^i.
      ii. *All* parties receive the output bit γ on the second output wire. They denote the output bit by γ.
      iii. S_2 and S_3 *only* receive the output bit on the third output wire. S_2 and S_3 denote this output by μ_2^i.
      iv. S_2 *only* receives the output bit on the fourth output wire. S_2 denotes this output by ν_2^i.
   (c) If any party receives output bit $\gamma = 0$, it aborts.
3. S_1 outputs $\mu_1 = \mu_1^1, \ldots, \mu_1^\ell$ and $\nu_1 = \nu_1^1, \ldots, \nu_1^\ell$.
4. S_2 outputs $\mu_2 = \mu_2^1, \ldots, \mu_2^\ell$ and $\nu_2 = \nu_2^1, \ldots, \nu_2^\ell$.
5. S_3 outputs $\mu_2 = \mu_2^1, \ldots, \mu_2^\ell$ and $\nu_1 = \nu_1^1, \ldots, \nu_1^\ell$.

Figure 7. The Protocol : Π_{Round}

The parties using the 3-party garbled circuit protocol from earlier securely compute the application of KMAC (with diversifer D) on the values $\mu = \mu_1 \oplus \mu_2 = \nu_1 \oplus \nu_2$ as follows. In addition party S_1 enters a random value $b_i^{(1)} \in \mathbb{F}_q$ for each output bit.

Padding: The parties first defin: $\bar{\mu}_1 = \mu_1 || 1 || 0^{r-|\mu_1|-2} || 1$, $\bar{\mu}_2 = \mu_2 || 0^{r-|\mu_2|}$, $\bar{\nu}_1 = \nu_1 || 1 || 0^{r-|\nu_1|-2} || 1$ and $\bar{\nu}_2 = \nu_2 || 0^{r-|\nu_2|}$ where $r = 1088$.
The parties then run the following circuit:

1. Compute $u = \bar{\mu}_1 \oplus \bar{\mu}_2$.
2. Compute $v = \bar{\nu}_1 \oplus \bar{\nu}_2$.
3. Check that $u = v$, if not, abort. i.e. output 1 on a special abort wire and 0 on all others; if no abort, then the abort wire should hold 0.
4. Compute KMAC(u): We need one absorbtion round, and for the squeezing we repeatedly apply the garbled round function for SHA-3, so as to obtain $3 \cdot 40 \cdot N$ bits of shared output. Thus we need to execute $3 \cdot 40 \cdot N/r$ rounds of squeezing, when $N = 1024$ and $r = 1088$, so we require 113 rounds of squeezing.
   Specifically: Let $p(x, y)$ denote the permutation function of SHA-3, with domain size of 1600 bits
   (a) Define: $R_0 = 0^{1088}$, $C_0 = 0^{512}$
   (b) $(R'_0, C'_0) = p(u, 0^{512})$
   (c) For $i = 1$ to 113: $(R'_i, C'_i) = p(R'_{i-1}, C'_{i-1})$.
   (d) The values $(R'_1, \ldots, R'_{113})$ are then used (combined with the respected $b_i^{(1)}$ input from S_1) to produce the output for S_2. In particular if bit k of R'_j corresponds to output bit b_i then we set $b_i^{(2)}$ to $b_i - b_i^{(1)} \pmod q$.

Figure 8. Protocol to securely evaluate KMAC on shared inputs

1. Using Figure 8 party S_1 and S_2 obtain an additive sharing over $\mathbb{F}_q$ of the output bits from the KMAC operation applied to $\mu = \mu_1 \oplus \mu_2 = \nu_1 \oplus \nu_2$.
2. Parties S_1 and S_2 now locally compute additive sharings modulo q of the random Gaussian values v_i, e_i and d_i from the Enc-Sub-CPA algorithm of LIMA. This is done by locally applying the algorithm to produce Gaussian values, GenerateGaussianNoise since it is a linear function of the input random bits. Thus party S_1 obtains three ring elements $(v^{(1)}, e^{(1)}, d^{(1)})$, and party S_2 obtains three ring elements $(v^{(2)}, e^{(2)}, d^{(2)})$ such that $v = v^{(1)} + v^{(2)}$, $e = e^{(1)} + e^{(2)}$ and $d = d^{(1)} + d^{(2)}$.
3. Party S_1 locally computes the values $\mathbf{v}^{(1)} = \mathsf{FFT}(v^{(1)})$, $\mathbf{e}^{(1)} = \mathsf{FFT}(e^{(1)})$, $s^{(1)} = \mathsf{FFT}^{-1}(\mathbf{b} \otimes \mathbf{v}^{(1)})$, $\mathbf{c}_1^{(1)} = (\mathbf{a} \otimes \mathbf{v}^{(1)}) \oplus \mathbf{e}^{(1)}$.
4. Party S_2 locally computes the values $\mathbf{v}^{(2)} = \mathsf{FFT}(v^{(2)})$, $\mathbf{e}^{(2)} = \mathsf{FFT}(e^{(2)})$, $s^{(2)} = \mathsf{FFT}^{-1}(\mathbf{b} \otimes \mathbf{v}^{(2)})$, $\mathbf{c}_1^{(2)} = (\mathbf{a} \otimes \mathbf{v}^{(2)}) \oplus \mathbf{e}^{(2)}$.
5. Parties S_1 and S_2 locally compute sharing of $y = y^{(1)} + y^{(2)} = s + d$ by computing $y^{(1)} = s^{(1)} + d^{(1)}$ and $y^{(2)} = s^{(2)} + d^{(2)}$.
6. The parties S_1 and S_2 open $\mathbf{c}_1^{(1)}$ and $\mathbf{c}_1^{(2)}$ to each other, so they can obtain $\mathbf{c}'_1$.
7. With private input $(y^{(1)}, \mu_1)$ and $(y^{2)}, \mu_2)$ from S_1 and S_2 the three parties compute (using a secure Garbled Circuit based secure computation protocol) the value

$$c'_0 = \mathsf{Trunc}\left(y^{(1)} + y^{(2)} + \Delta_q \cdot \mathsf{BV\text{-}2\text{-}RE}(\mu_1 \oplus \mu_2), \ell\right).$$

   with S_1 and S_2 obtaining the output.

Figure 9. Protocol to securely evaluate $\mathsf{Enc\text{-}CPA\text{-}Sub}((\mu_1, \nu_1), (\mu_1, \nu_2), (\mu_2, \nu_1), \mathfrak{pk}, D)$

1. Let ℓ denote the length of c_0, abort if $\ell \neq 0 \pmod 8$.
2. Party S_1 locally computes
$$f_1^{1,2} \leftarrow c_0 - \mathsf{Trunc}(\mathsf{FFT}^{-1}(\mathbf{s}_1^{1,2} \otimes \mathbf{c}_1), \ell) \text{ and } f_1^{1,3} \leftarrow c_0 - \mathsf{Trunc}(\mathsf{FFT}^{-1}(\mathbf{s}_1^{1,3} \otimes \mathbf{c}_1), \ell).$$
3. Party S_2 locally computes
$$f_2^{1,2} \leftarrow \mathsf{Trunc}(\mathsf{FFT}^{-1}(\mathbf{s}_2^{1,2} \otimes \mathbf{c}_1), \ell) \text{ and } f_1^{2,3} \leftarrow c_0 - \mathsf{Trunc}(\mathsf{FFT}^{-1}(\mathbf{s}_1^{2,3} \otimes \mathbf{c}_1), \ell).$$
4. Party S_3 locally computes
$$f_2^{1,3} \leftarrow \mathsf{Trunc}(\mathsf{FFT}^{-1}(\mathbf{s}_2^{1,3} \otimes \mathbf{c}_1), \ell) \text{ and } f_2^{2,3} \leftarrow \mathsf{Trunc}(\mathsf{FFT}^{-1}(\mathbf{s}_2^{2,3} \otimes \mathbf{c}_1), \ell).$$
5. Note that after these operations we have $f_1^{1,2} + f_2^{1,2} = f_1^{1,3} + f_2^{1,3} = f_1^{2,3} + f_2^{2,3} = c_0 - \mathsf{Trunc}(\mathsf{FFT}^{-1}(\mathbf{s} \otimes \mathbf{c}_1), \ell)$.
6. S_1, S_2 and S_3 securely compute $\mathsf{Round}\left((f_1^{1,2}, f_1^{1,3}), (f_2^{1,2}, f_1^{2,3}), (f_2^{1,3}, f_2^{2,3}), \ell\right)$, as given above in Figure 7. Denote the output of S_1 from this computation by μ_1, ν_1, the output of S_2 by μ_2, ν_2, and the output of S_3 by μ_2, ν_1, with $\mu_1, \nu_1, \mu_2, \nu_2 \in \{0,1\}^\ell$. If any party received an abort in the Round function computation, then it does not proceed.
7. The parties now run $\mathsf{Enc\text{-}CPA\text{-}Sub}(\mu_1, \mu_2, \nu_1, \nu_2, \mathfrak{pk}, 0x03)$ using the protocol in Figure 9, so that parties S_1 and S_2 obtain c_0' and $\mathbf{c}_1'$.
8. If S_1 or S_2 detect that $c_0 \neq c_0'$ or $\mathbf{c}_1 \neq \mathbf{c}_1'$ then the parties abort.
9. All three parties reveal the first $\ell - 256$ bits of μ_1, μ_2, ν_1 and ν_2 to each other using a broadcast channel (ensuring a sending party cannot send different values to different players). Call these values μ_1', μ_2', ν_1' and ν_2'. A party now aborts if any value it receives, which it owns, does not equal what they expected, i.e. S_1 aborts if ν_1' that they hold does not equal the ν_1' that was sent from party S_3.
10. The parties compute $m = \mu_1' \oplus \mu_2'$ and $m' = \nu_1' \oplus \nu_2'$ and abort if $m \neq m'$.
11. The parties output m.

Figure 10. Secure Evaluation of $\mathsf{Dec\text{-}CCA}(\mathbf{c}, \mathfrak{sk})$

Security of the protocol: The Round function is computed by a protocol that is secure against malicious adversaries. Intuitively, this means that the view of each party can be trivially simulated since S_1, S_2 and S_3 receive nothing but random shares as output from this subprotocol and no other messages are even sent. However, the parties *can* provide incorrect values at all steps of the protocol. Specifically, a corrupt S_1 can input an incorrect f_1 into Round, similarly a corrupt S_2 can input an incorrect f_2 into Round, etc. We resolve this problem by adding redundancy into the computation.

First, we compute the initial f values three times; once between each different pair or parties. Since at least one of these pairs is guaranteed to be honest, we have that the output of the operation $c_0 - \mathsf{Trunc}(\mathsf{FFT}^{-1}(\mathbf{s} \otimes \mathbf{c}_1))$ will be correct for this pair. Since the Round function computation verifies that all these values are equal (or else $\gamma = 0$ in the output and the parties learn nothing and abort), we have that this must be correct.

The parties then the protocol to evaluate Enc-CPA-Sub from Figure 9 to obtain the re-encryption $(c_0', \mathbf{c}_1')$. Note, that for this to be correct, and so the equality check to pass, the servers must act honestly. Otherwise an invalid ciphertext is produced. Finally, the output message is obtained, and checked for correctness, using the redunancy inherent in the output of the Round function.

We prove the security of the protocol via simulation by constructing a simulator $\mathcal{S}$ for the adversary $\mathcal{A}$. The simulator $\mathcal{S}$ knows the shares of the key held by each party, and works as follows:

– If S_1 is corrupted by $\mathcal{A}$, then the simulator $\mathcal{S}$ sends $c_0, \mathbf{c}_1$ to the trusted party computing the functionality and receives back $m = x \| s$. Then, $\mathcal{S}$ invokes $\mathcal{A}$ and receives the inputs $(f_1^{1,2}, f_1^{1,3})$ that $\mathcal{A}$ inputs to the Round function. Since $\mathcal{S}$ knows $\mathbf{s}_1^{1,2}$ and $\mathbf{s}_1^{1,3}$, it can verify if $\mathcal{A}$ computed these correctly. If not, then $\mathcal{S}$ sends $\perp$ to the trusted party, simulates the output of Round providing $\gamma = 0$ at the appropriate places and halts. (All other outputs of Round are given as random.) Else, $\mathcal{S}$ provides output of Round to be $\gamma = 1$ and the μ^2, ν^2 values as random.
The simulation of the application of KMAC via a Garbled Circuit can be done, assuming an ideal functionality for the secure computation of KMAC, in the standard way. Note, that if S_1 lies about its input into the KMAC algorithm, or lies about its value $(y^{(1)}, \mu_1)$ input into the Trunc evaluation, then with overwhelming probability party S_2 will abort when checking $c_0' = c_1'$ or $\mathbf{c}_1' = \mathbf{c}_1$.

The view of $\mathcal{A}$ in the simulation is clearly identical to its view in a hybrid execution where the function Round, Trunc and a function to perform secure computation are ideal functionalities. Thus, the output distributions are computationally indistinguishable, as required.

– The simulation for S_2 and S_3 is similar; with S_3 being a little simpler.

4.4 Secure Evaluation of Decap-CCA$(\mathbf{c}, \mathfrak{sk})$

The distributed decapsulation routine works very much like the distributed decryption routine. However, to obtain the same ideal functionality of a real decapsulation routine we need to evaluate the KDF computation within a secure computation. This can be done using the same method we use to evaluate the KMAC needed in the XOF computation; since in LIMA both are based on different modes of the SHA-3 based KMAC operation. The overal protocol is given in Figure 11.

1. Let ℓ denote the length of c_0, abort if $\ell \neq 0 \pmod 8$.
2. Party S_1 computes
$$f_1^{1,2} \leftarrow c_0 - \mathsf{Trunc}(\mathsf{FFT}^{-1}(\mathbf{s}_1^{1,2} \otimes \mathbf{c}_1), \ell) \text{ and } f_1^{1,3} \leftarrow c_0 - \mathsf{Trunc}(\mathsf{FFT}^{-1}(\mathbf{s}_1^{1,3} \otimes \mathbf{c}_1), \ell).$$
3. Party S_2 computes
$$f_2^{1,2} \leftarrow \mathsf{Trunc}(\mathsf{FFT}^{-1}(\mathbf{s}_2^{1,2} \otimes \mathbf{c}_1), \ell) \text{ and } f_1^{2,3} \leftarrow c_0 - \mathsf{Trunc}(\mathsf{FFT}^{-1}(\mathbf{s}_1^{2,3} \otimes \mathbf{c}_1), \ell).$$
4. Party S_3 computes
$$f_2^{1,3} \leftarrow \mathsf{Trunc}(\mathsf{FFT}^{-1}(\mathbf{s}_2^{1,3} \otimes \mathbf{c}_1), \ell) \text{ and } f_2^{2,3} \leftarrow \mathsf{Trunc}(\mathsf{FFT}^{-1}(\mathbf{s}_2^{2,3} \otimes \mathbf{c}_1), \ell).$$
5. Note that after these operations we have $f_1^{1,2} + f_2^{1,2} = f_1^{1,3} + f_2^{1,3} = f_1^{2,3} + f_2^{2,3} = c_0 - \mathsf{Trunc}(\mathsf{FFT}^{-1}(\mathbf{s} \otimes \mathbf{c}_1), \ell)$.
6. S_1, S_2 and S_3 securely compute Round $\big((f_1^{1,2}, f_1^{1,3}), (f_2^{1,2}, f_1^{2,3}), (f_2^{1,3}, f_2^{2,3}), \ell\big)$, as given above in Section 4.1. Denote the output of S_1 from this computation by μ_1, ν_1, the output of S_2 by μ_2, ν_2, and the output of S_3 by μ_2, ν_1 (recall that S_3 receives the same μ_2 as S_2 and the same ν_1 as S_1). Note that $\mu_1, \nu_1, \mu_2, \nu_2 \in \{0,1\}^\ell$. If any party received an abort in the Round function computation, then it does not proceed.
7. The parties now run Enc-Sub-CPA$(\mu_1, \mu_2, \nu_1, \nu_2, \mathfrak{pk}, 0x03)$ so that parties S_1 and S_2 obtain c_0' and $\mathbf{c}_1'$.
8. If S_1 or S_2 detect that $c_0 \neq c_0'$ or $\mathbf{c}_1 \neq \mathbf{c}_1'$ then the parties abort.
9. Using a method similar to that above for producing the KMAC value in Enc-Sub-CPA the parties now compute the value
$$\mathbf{k} \leftarrow \mathsf{KDF}^{[\ell]}(\mu_1 \oplus \mu_2) = \mathsf{KDF}^{[\ell]}(\nu_1 \oplus \nu_2)$$
using their private inputs $\mu_1, \mu_2, \nu_1, \nu_2$. Note, we again need to check for consistency of inputs within the garbled circuit.
10. The value $\mathbf{k}$ is returned.

Figure 11. Secure Evaluation of Decap-CCA$(\mathbf{c}, \mathfrak{sk})$

4.5 Experimental Results

Using the basic LIMA parameters of $q = 40961$ and $N = 1024$ we implemented the above protocol, and run it on a set of three Linux RHEL servers running on AWS of type `t2.small` with 2GB of RAM. The total run time for distributed decyrption was 4280 milliseconds. The main cost was the need to perform the inital decryption, and then re-encrypt, without recovering the message in plaintext. This is inherent in the methodology adopted by LIMA, and many other of the PQC candidate algorithms, for producing a chosen ciphertext secure encryption algorithm. More MPC friendly methods to obtain CCA security could reduce this run time considerably, but that does not seem to have been a design goal for any of the candidate submissions. For distributed decapsulation in the KEM algorithm we achieved an execution time of 4342 milliseconds.

5 Distributed Key Generation for Ring-LWE Encryption

Distributed key generation can be performed relatively straightforwardly using a generic MPC system based on linear secret sharing which supports two-out-of-three threshold access structures and gives active security with abort. As explained earlier we selected SCALE-MAMBA to do this, as we could use an off-the-shelf system.

SecGauss()

1. $[a] \leftarrow 0$.
2. For $i \in [0, \ldots, 19]$ do
   (a) $[b] \leftarrow$ Bits.
   (b) $[b'] \leftarrow$ Bits.
   (c) $[a] \leftarrow [a] + [b] - [b']$.
3. Return $[a]$.

Figure 12. Securely Generating Approximate Gaussians

The main difficulty in key generation would appear to be the need to generate the approximate Gaussian distributions needed for LIMA. However, the specific distribution method chosen in LIMA dovetails nicely with the offline pre-processing found in SCALE-MAMBA. This results in the method to securely generate approximate Gaussian distributions given in Figure 12, which we note becomes a completely local operation in the online phase of the MPC protocol.

Inner($\underline{a}$)

1. Common input a polynomial $\underline{a} \in R_q$.
2. For $i \in [0, \ldots, N-1]$ do
   (a) $[\underline{s}]_i \leftarrow$ SecGauss().
   (b) $[\underline{e}]_i \leftarrow$ SecGauss().
   (c) $([\underline{s}_1^{1,2}]_i, [\underline{s}_1^{1,3}]_i, [c]) \leftarrow$ Triples.
   (d) $([\underline{s}_1^{2,3}]_i, [b], [c]) \leftarrow$ Triples.
   (e) $[\underline{s}_2^{1,2}]_i = [\underline{s}]_i - [\underline{s}_1^{1,2}]_i$.
   (f) $[\underline{s}_2^{1,3}]_i = [\underline{s}]_i - [\underline{s}_1^{1,3}]_i$.
   (g) $[\underline{s}_2^{2,3}]_i = [\underline{s}]_i - [\underline{s}_1^{2,3}]_i$.
   (h) Output-To$(1, [\underline{s}_1^{1,2}]_i)$.
   (i) Output-To$(1, [\underline{s}_1^{1,3}]_i)$.
   (j) Output-To$(2, [\underline{s}_1^{2,3}]_i)$.
   (k) Output-To$(2, [\underline{s}_2^{1,2}]_i)$.
   (l) Output-To$(3, [\underline{s}_2^{1,3}]_i)$.
   (m) Output-To$(3, [\underline{s}_2^{2,3}]_i)$.
3. $[\underline{b}] \leftarrow \underline{a} \cdot [\underline{s}] + [\underline{e}] \pmod{\Phi_{2 \cdot N}(X)}$.
   This is a completely local operation as $\underline{a}$ is public which we do as follows for $k = 0, \ldots, N-1$
   (a) $[\underline{b}]_k \leftarrow \left(\sum_{i+j=k} \underline{a}_i \cdot [\underline{s}]_j\right) - \left(\sum_{i+j=k+N} \underline{a}_i \cdot [\underline{s}]_j\right) + [\underline{e}]_k$.
4. For $i \in [0, \ldots, N-1]$ do
   (a) Output$([\underline{b}]_i)$.

Figure 13. Inner MPC Routine of KeyGen

From this it is easy to produce the key generation procedure which we give in terms of an inner MPC-core of the algorithm (which mainly consists of local operations and opening values to different players which is implemented in SCALE-MAMBA) given in Figure 13, plus non-interactive local operations which are purely about placing data into the correct formats given in Figure 14 We make extensive use of the fact that the FFT operation is linear. In our

algorithms we utilize vectors/polynomials of secret shared values which we will write as $[\underline{f}]$ which we use to represent the element in R given by $[\underline{f}]_0 + [\underline{f}]_1 \cdot X + \ldots + [\underline{f}]_{N-1} \cdot X^{N-1}$.

KeyGen()

1. All players agree on a key for a XOF XOF.
2. $\underline{a} \xleftarrow{\text{XOF}} \mathbb{F}_q^N$.
3. Inner($\underline{a}$).
4. $\mathbf{a} \leftarrow \text{FFT}(\underline{a})$.
5. $\mathbf{b} \leftarrow \text{FFT}(\underline{b})$.
6. $\mathfrak{pk} \leftarrow (\mathbf{a}, \mathbf{b})$.
7. Player S_1 executes $\mathbf{s}_1^{1,2} \leftarrow \text{FFT}(\underline{s}_1^{1,2})$.
8. Player S_1 executes $\mathbf{s}_1^{1,3} \leftarrow \text{FFT}(\underline{s}_1^{1,3})$.
9. Player S_2 executes $\mathbf{s}_2^{1,2} \leftarrow \text{FFT}(\underline{s}_2^{1,2})$.
10. Player S_2 executes $\mathbf{s}_1^{2,3} \leftarrow \text{FFT}(\underline{s}_1^{2,3})$.
11. Player S_3 executes $\mathbf{s}_2^{1,3} \leftarrow \text{FFT}(\underline{s}_2^{1,3})$.
12. Player S_3 executes $\mathbf{s}_2^{2,3} \leftarrow \text{FFT}(\underline{s}_2^{2,3})$.

Figure 14. Main Key Generation Routine

5.1 Experimental Results

We implemented the above key generation phase within the SCALE-MAMBA framework for the parameters $N = 1024$ and $q = 40961$ of LIMA. We used the settings of Shamir secret sharing and the Maurer [23] based offline settings of SCALE-MAMBA. Our experiments for this component were executed on three Linux Ubuntu machines with Intel i7-7700K processors running at 4.20 GHz, and with 8192KB cache and 32GB RAM.

The SCALE-MAMBA system runs in an integrated offline–online manner, however one can program it so as to obtain estimates for the execution times of both the offline and the online phases. An important parameter within the system is a statistical security parameter secp which defines the probability that an adversary can get the pre-processing to invalid data.

The variable $s =$`sacrifice_stat_sec` in the system defines the value of secp via the equation

$$\text{secp} = \lceil \log_2 q \rceil \cdot \left\lceil \frac{s}{\lceil \log_2 q \rceil} \right\rceil.$$

When q is large (i.e. $q > 2^{128}$), as it is in most envisioned executions of SCALE-MAMBA the default value of $s = 40$ results in a suitable security parameter. However, for our small value of q we needed to modify s so as to obtain a suitable value of secp. We note, that this setting only affects the runtime for the offline phase; and as can be seen the effect on the run times in Table 1 is marginal.

The online run time takes 1.22 seconds, although roughly one second of this is used in performing the 6144 output operations. On further investigation we found this was because SCALE-MAMBA performs all the reveals in a sequential as opposed to batch manner (requiring 6144 rounds of communication as opposed to one). We suspect a more careful tuned implementation could reduce the online time down to less than a quarter of a second. However, our implementation of the Key Generation method in SCALE-MAMBA took about a day of programmers time; thus using a general system (even if inefficient) can be more efficient on the development time.

Acknowledgements

This work has been supported in part by ERC Advanced Grant ERC-2015-AdG-IMPaCT and by the Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under contract No. N66001-15-C-4070.

s	secp	Time (Seconds)
40	48	20.2
80	80	20.7
128	128	23.1

Table 1. Times to produce the offline data for the Key Generation operation. This is essentially producing 81920 shared random bits, 2048 multiplication triples and enough shared randomness to enable the output of the shared keys.

References

1. Albrecht, M.R., Lindell, Y., Orsini, E., Osheter, V., Paterson, K.G., Peer, G., Smart, N.P.: LIMA-1.1 : A PQC encryption scheme (2018), https://lima-pq.github.io/
2. Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Post-quantum key exchange - A New Hope. In: Holz, T., Savage, S. (eds.) 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016. pp. 327–343. USENIX Association (2016), https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/alkim
3. Aly, A., Keller, M., Orsini, E., aru, D.R., Scholl, P., Smart, N.P., Wood, T.: SCALE and MAMBA documentation (2018), https://homes.esat.kuleuven.be/~nsmart/SCALE/
4. Asharov, G., Jain, A., López-Alt, A., Tromer, E., Vaikuntanathan, V., Wichs, D.: Multiparty computation with low communication, computation and interaction via threshold FHE. In: Pointcheval, D., Johansson, T. (eds.) Advances in Cryptology – EUROCRYPT 2012. Lecture Notes in Computer Science, vol. 7237, pp. 483–501. Springer, Heidelberg, Germany, Cambridge, UK (Apr 15–19, 2012)
5. Beaver, D., Micali, S., Rogaway, P.: The round complexity of secure protocols (extended abstract). In: 22nd Annual ACM Symposium on Theory of Computing. pp. 503–513. ACM Press, Baltimore, MD, USA (May 14–16, 1990)
6. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. In: Goldwasser, S. (ed.) ITCS 2012: 3rd Innovations in Theoretical Computer Science. pp. 309–325. Association for Computing Machinery, Cambridge, MA, USA (Jan 8–10, 2012)
7. Brandao, L.T.A.N., Mouha, N., Vassilev, A.: Threshold schemes for cryptographic primitives: Challenges and opportunities in standardization and validation of threshold cryptography (2018), https://csrc.nist.gov/publications/detail/nistir/8214/draft
8. Choudhury, A., Loftus, J., Orsini, E., Patra, A., Smart, N.P.: Between a rock and a hard place: Interpolating between MPC and FHE. In: Sako, K., Sarkar, P. (eds.) Advances in Cryptology – ASIACRYPT 2013, Part II. Lecture Notes in Computer Science, vol. 8270, pp. 221–240. Springer, Heidelberg, Germany, Bengalore, India (Dec 1–5, 2013)
9. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology – CRYPTO 2012. Lecture Notes in Computer Science, vol. 7417, pp. 643–662. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 19–23, 2012)
10. Desmedt, Y.: Society and group oriented cryptography: A new concept. In: Pomerance, C. (ed.) Advances in Cryptology – CRYPTO'87. Lecture Notes in Computer Science, vol. 293, pp. 120–127. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 16–20, 1988)
11. Desmedt, Y., Frankel, Y.: Threshold cryptosystems. In: Brassard, G. (ed.) Advances in Cryptology – CRYPTO'89. Lecture Notes in Computer Science, vol. 435, pp. 307–315. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 20–24, 1990)
12. Fouque, P.A., Pointcheval, D.: Threshold cryptosystems secure against chosen-ciphertext attacks. In: Boyd, C. (ed.) Advances in Cryptology – ASIACRYPT 2001. Lecture Notes in Computer Science, vol. 2248, pp. 351–368. Springer, Heidelberg, Germany, Gold Coast, Australia (Dec 9–13, 2001)
13. Frederiksen, T.K., Lindell, Y., Osheter, V., Pinkas, B.: Fast distributed RSA key generation for semi-honest and malicious adversaries. In: Shacham, H., Boldyreva, A. (eds.) Advances in Cryptology – CRYPTO 2018, Part II. Lecture Notes in Computer Science, vol. 10992, pp. 331–361. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 19–23, 2018)
14. Gentry, C.: A fully homomorphic encryption scheme. Ph.D. thesis, Stanford University (2009), http://crypto.stanford.edu/craig
15. Hazay, C., Mikkelsen, G.L., Rabin, T., Toft, T.: Efficient RSA key generation and threshold Paillier in the two-party setting. In: Dunkelman, O. (ed.) Topics in Cryptology – CT-RSA 2012. Lecture Notes in Computer Science, vol. 7178, pp. 313–331. Springer, Heidelberg, Germany, San Francisco, CA, USA (Feb 27 – Mar 2, 2012)
16. Ito, M., Nishizeki, T., Saito, A.: Secret sharing scheme realizing general access structure. Electronics and Communications in Japan (Part III: Fundamental Electronic Science) 72, 56–64 (1989)

17. Katz, J., Yung, M.: Threshold cryptosystems based on factoring. In: Zheng, Y. (ed.) Advances in Cryptology – ASIACRYPT 2002. Lecture Notes in Computer Science, vol. 2501, pp. 192–205. Springer, Heidelberg, Germany, Queenstown, New Zealand (Dec 1–5, 2002)
18. Keller, M., Pastro, V., Rotaru, D.: Overdrive: Making SPDZ great again. In: Nielsen, J.B., Rijmen, V. (eds.) Advances in Cryptology – EUROCRYPT 2018, Part III. Lecture Notes in Computer Science, vol. 10822, pp. 158–189. Springer, Heidelberg, Germany, Tel Aviv, Israel (Apr 29 – May 3, 2018)
19. Keller, M., Rotaru, D., Smart, N.P., Wood, T.: Reducing communication channels in MPC. In: Catalano, D., De Prisco, R. (eds.) SCN 18: 11th International Conference on Security in Communication Networks. Lecture Notes in Computer Science, vol. 11035, pp. 181–199. Springer, Heidelberg, Germany, Amalfi, Italy (Sep 5–7, 2018)
20. Libert, B., Yung, M.: Non-interactive CCA-secure threshold cryptosystems with adaptive security: New framework and constructions. In: Cramer, R. (ed.) TCC 2012: 9th Theory of Cryptography Conference. Lecture Notes in Computer Science, vol. 7194, pp. 75–93. Springer, Heidelberg, Germany, Taormina, Sicily, Italy (Mar 19–21, 2012)
21. Lindell, Y.: Fast secure two-party ECDSA signing. In: Katz, J., Shacham, H. (eds.) Advances in Cryptology – CRYPTO 2017, Part II. Lecture Notes in Computer Science, vol. 10402, pp. 613–644. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 20–24, 2017)
22. Lindell, Y., Smart, N.P., Soria-Vazquez, E.: More efficient constant-round multi-party computation from BMR and SHE. In: Hirt, M., Smith, A.D. (eds.) TCC 2016-B: 14th Theory of Cryptography Conference, Part I. Lecture Notes in Computer Science, vol. 9985, pp. 554–581. Springer, Heidelberg, Germany, Beijing, China (Oct 31 – Nov 3, 2016)
23. Maurer, U.: Secure multi-party computation made simple. Discrete Applied Mathematics 154(2), 370–381 (2006)
24. Mohassel, P., Rosulek, M., Zhang, Y.: Fast and secure three-party computation: The garbled circuit approach. In: Ray, I., Li, N., Kruegel:, C. (eds.) ACM CCS 15: 22nd Conference on Computer and Communications Security. pp. 591–602. ACM Press, Denver, CO, USA (Oct 12–16, 2015)
25. NIST National Institute for Standards and Technology: SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash (2016), http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-185.pdf
26. Shoup, V.: Practical threshold signatures. In: Preneel, B. (ed.) Advances in Cryptology – EUROCRYPT 2000. Lecture Notes in Computer Science, vol. 1807, pp. 207–220. Springer, Heidelberg, Germany, Bruges, Belgium (May 14–18, 2000)
27. Shoup, V., Gennaro, R.: Securing threshold cryptosystems against chosen ciphertext attack. In: Nyberg, K. (ed.) Advances in Cryptology – EUROCRYPT'98. Lecture Notes in Computer Science, vol. 1403, pp. 1–16. Springer, Heidelberg, Germany, Espoo, Finland (May 31 – Jun 4, 1998)
28. Smart, N.P., Wood, T.: Error-detecting in monotone span programs with application to communication efficient multi-party computation. Cryptology ePrint Archive, Report 2018/467 (2018), https://eprint.iacr.org/2018/467
29. Songhori, E.M., Hussain, S.U., Sadeghi, A.R., Schneider, T., Koushanfar, F.: TinyGarble: Highly compressed and scalable sequential garbled circuits. In: 2015 IEEE Symposium on Security and Privacy. pp. 411–428. IEEE Computer Society Press, San Jose, CA, USA (May 17–21, 2015)