Zaphod: Efficiently Combining LSSS and Garbled Circuits in SCALE

Abdelrahaman Aly[1], Emmanuela Orsini[1], Dragos Rotaru[1,2], Nigel P. Smart[1,2], and Tim Wood[1,2]

[1] imec-COSIC, KU Leuven, Leuven, Belgium.
[2] University of Bristol, Bristol, UK.
abdelrahaman.aly@esat.kuleuven.be,emmanuela.orsini@kuleuven.be,
dragos.rotaru@esat.kuleuven.be,nigel.smart@kuleuven.be,t.wood@kuleuven.be

Abstract. We present modifications to the MPC system `SCALE-MAMBA` to enable the evaluation of garbled circuit (GC) based MPC functionalities and Linear Secret Sharing (LSSS) based MPC functionalities along side each other. This allows the user to switch between different MPC paradigms to achieve the best performance. To do this we present modifications to the GC-based MPC protocol of Hazay et al. (Asiacrypt 2017) (to enable it to support reactive computation), and combine different aspects of their pre-processing phase with those of Wang et al. (CCS 2017), in order to optimize our pre-processing protocols. We also give a more efficient method for producing daBits (double authenticated Bits) than that presented in the work of Rotaru and Wood (ePrint 2019). Finally, we examine how the functionality can be integrated within the existing MPC framework `SCALE-MAMBA`.

1 Introduction

Multi-Party Computation (MPC) is a set of techniques that enable different parties to compute a function on their joint (and privately held) data, without disclosing anything about the data itself, bar what can be derived from the result of the function. It was first introduced in the 1980s and it has, over the last ten years, seen a remarkable improvement in terms of performance. Indeed, MPC has reached a level where companies are starting commercial deployments of MPC based solutions.

MPC comes in essentially two flavours (paradigms). One flavour operates via linear secret-sharing schemes (LSSSs) over some finite field $\mathbb{F}_p$, or sometimes a finite ring. The function to be computed is expressed as an arithmetic circuit (this is not quite true, but this abstraction is good enough for us). Secure computation proceeds by evaluating the circuit via a series of messages being exchanged. We denominate each exchange a *communication round*. The number of rounds depends on the multiplicative depth of the circuit, with the total amount of communication (volume) depending on the number of multiplication gates. Additions do not require message exchanges, hence we consider them to be "free" in terms of communication. Protocols in this family date back to [BGW88, CCD88], with modern efficient protocols (for various access structures) including those in [BDOZ11, DPSZ12, CGH+18, SW19].

The second flavour operates via so-called garbled circuits (GC). In the GC paradigm the function to be evaluated is expressed as a boolean combinatorial circuit. Each AND-gate in the circuit is "encrypted" via a garbled table. In the first phase of computation the parties produce these encrypted tables, then in the second phase the parties evaluate the function using the tables. In this case, the total amount of communication depends on the number of AND gates, but the round complexity is constant. In the passively secure two party case this idea goes back to Yao [Yao82], which was generalised to the honest-majority multi-party setting by Beaver et al. [BMR90]. Recently, there has been renewed interest in n-party actively-secure protocols in this paradigm, such as [HSS17, WRK17]. The focus in these newer protocols is almost always on full-threshold access structures.

It is clear that these two flavours of MPC are mutually complementary, i.e. evaluation of some functions is more suited to the LSSS paradigm, and for others it is more suited to the GC paradigm. For example, we would prefer to use boolean circuits to perform computation over the integers where the operation is better expressed as a boolean circuit, and arithmetic circuits where the computation to be performed is best expressed as an arithmetic circuit. Thus we want to combine $\mathbb{Z}_p$ operations and **B**oolean operations, and indeed this is the goal of this work.

We call our system for doing this Zaphod, after the two headed character **Z**a**p**hod **B**eeblebrox from Hitchhiker's Guide to the Galaxy.

Let us consider the following example, where a set of parties wants to compute the function $F : \mathbb{F}_p^{n+m} \to \{0,1\}$ defined by

$$F(x_1, \ldots, x_n, y_1, \ldots, y_m) := \left(\sum_{i=1}^{n} x_i^2\right) \overset{?}{>} \left(\sum_{i=1}^{m} y_i^2\right).$$

In this case, we might perform the summations of squares using arithmetic circuits over a field $\mathbb{F}_p$, where p is large enough prime to ensure that the summation is valid, i.e. no overflow occurs. We might nonetheless prefer to perform the comparison using a boolean circuit, since comparison is hard in the LSSS paradigm, but integer addition is free in the LSSS paradigm and integer multiplication is also cheaper in the LSSS paradigm.

However, a major drawback has been that almost all systems have required one to choose a specific paradigm for one's MPC computations. While significant work has been done on rectifying this in the two-party setting with passive security [BPSW07, HKS+10, KSS14, DSZ15, CGR+17, BDK+18] and some work in the three-party honest-majority active security setting [MR18], such protocols are often tailored to specific choices of GC and LSSS protocols.

By contrast, Rotaru and Wood [RW19] recently introduced a mechanism, called daBits (for doubly-authenticated Bits), to combine the GC and LSSS paradigms for general actively secure n-party computation. The idea there is to make use of the GC and LSSS protocols as black boxes to the fullest possible extent to allow greater flexibility in choosing the two protocols. This is done by the parties generating bits which are authenticated both in the GC and LSSS world, and then using these bits to transfer data from one world to the other, and then back again. Whilst this is good for general purpose computation, there are significant engineering issues to be addressed to apply this idea within a working and practical system. In particular, how such constructs are to be exposed to the user of the system? What data-types one will support on either side? How the system will be integrated?

Our Contribution In this paper we answer these questions, and explain an integration of this idea into the LSSS-based `SCALE-MAMBA` system [AKO+18]. `SCALE-MAMBA` implements state of the art MPC protocols over finite fields $\mathbb{F}_p$, for various access structures. A number of issues need to be addressed in such an endeavour:

- `SCALE-MAMBA` supports reactive computation (i.e. data can be processed so that the system finishes a specific computation to obtain an intermediate disclosed result, before continuing with another secure computation using the same data). For this, we need to modify existing GC based MPC protocols to also support reactive computation. This in turn influences our selection of the underlying n-party GC protocol. Thus, in this work, we adapt the BMR-style protocol of [HSS17]. The reason for selecting [HSS17] over, say [WRK17] is that the former is role symmetric and it is easier to turn into a reactive MPC protocol compared to [WRK17].
- The GC-based protocol of [HSS17] makes use pre-processing of so-called authenticated bits and authenticated AND-triples, via OT-extension. There are a number of different ways of producing such triples e.g., Oblivious Transfer (OT). Thus we need to select the ones which are most efficient from the existing literature.
- The original method of generating daBits was very costly, involving expensive cut-and-choose and bucketing techniques. In this work we give a method that deals explicitly with shares of secrets (rather than handles as described in [RW19]), which allows the generation of daBits with total communication cost that is up to one quarter less and with a throughput of up to twice as much.
- `SCALE-MAMBA` comes with a programming language which enables a user to easily program the system. Producing boolean circuit level abstractions is a cumbersome task. Instead of users producing circuit descriptions of functions, it is more desirable to have a suitable high level abstraction in which a user can program her functions on the GC side of the computation. For this, we allow the user to interact with GC computations via an abstraction of signed 64-bit integers, much like any high level programming language would do, for access to real circuits in silicon.

Despite the LSSS operations being relatively general in regard to which access structure they support, the GC operations are set to be secure for full threshold access structures only. Our overall protocol is to be secure for the access structure supported by the underlying LSSS, assuming output to parties is always performed in the LSSS scheme.

Paper Outline In Figure 1, we show the relationship between our functionalities. Section 2, after an introduction of some notation from previous papers, contains our online functionality $\mathcal{F}_{\mathsf{MPC}}$. Section 3 describes our pre-processing functionality $\mathcal{F}_{\mathsf{Prep}}$ that provides the correlated randomness for both the LSSS ($\mathcal{F}_{\mathsf{Prep}}[\mathsf{SPDZ}]$) and BMR ($\mathcal{F}_{\mathsf{Prep}}[\mathsf{BMR}]$) online evaluation, and the doubly authenticated bits (daBits) to be used to convert from one paradigm to the other. Then in Section 4 we provide a new, more efficient method to produce daBits than the method in [RW19]. In Section 5 we give the online protocol, describing how we use the daBits to produce the conversion between the LSSS computations and the GC computations. Then in Section 6 we discuss how all this is integrated together into a single system, and how different aspects of the MPC computation and pre-processing are balanced between each other.

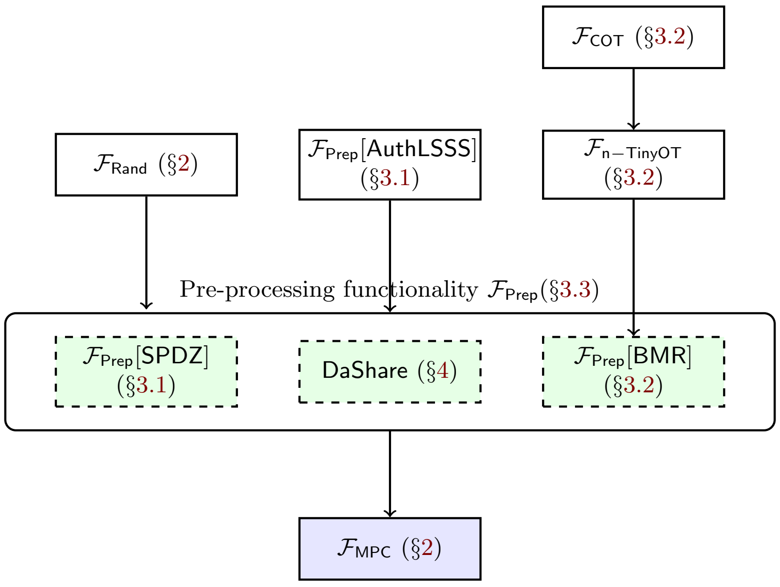
$\mathcal{F}_{\mathsf{COT}}$ (§3.2)
$\mathcal{F}_{\mathsf{Rand}}$ (§2)
$\mathcal{F}_{\mathsf{Prep}}$[AuthLSSS] (§3.1)
$\mathcal{F}_{\mathsf{n-TinyOT}}$ (§3.2)
Pre-processing functionality $\mathcal{F}_{\mathsf{Prep}}$(§3.3)
$\mathcal{F}_{\mathsf{Prep}}$[SPDZ] (§3.1)
DaShare (§4)
$\mathcal{F}_{\mathsf{Prep}}$[BMR] (§3.2)
$\mathcal{F}_{\mathsf{MPC}}$ (§2)

Fig. 1. Our functionalities

2 Preliminaries

Here we define the basic notation and constructs needed to describe our protocols. First, we explain the abstract LSSS-based computation which we will support; here we are totally general and allow n-party secure computation over a number of access structures (all of which are supported in the `SCALE-MAMBA` system). We also specify our notation for authenticated secret-shared values on the $\mathbb{F}_p$-side of the computation. Then, we introduce the equivalent authenticated sharing for bits on the GC side of the computation. We conclude this section by giving the abstract functionality which our modification to `SCALE-MAMBA` aims to achieve.

Notation We let κ (resp. sec) denote the computational (resp. statistical) security paramenter and $\mathcal{P} = \{P_1, \ldots, P_n\}$ be the set of parties. In practice we set $\kappa = 128$ (so as to use AES in our PRF) and, in our experiments we utilized $\mathsf{sec} = 40$ and $\mathsf{sec} = 80$. We denote by $[d]$ the set of integers $\{1, \ldots, d\}$ and use bold lower-case letters for vectors of elements over both binary and arithmetic fields.

The protocols presented in this work are all secure in the Universal Composability framework of Canetti [Can00]. We consider security against a static, malicious adversary. For the most part, the protocols presented here are secure for any full-threshold or Q2 access structure. In some of our protocols we will need a coin-tossing functionality $\mathcal{F}_{\mathsf{Rand}}$, which given a set $\mathcal{D}$, outputs a uniformly random element r from $\mathcal{D}$.

2.1 LSSS-Based MPC for Arithmetic Circuits

In this section we detail the type of secret-sharing-based MPC we will be utilizing, and the associated access structures.

We let $\langle \cdot \rangle_p$ denote a linear secret sharing scheme (LSSS) over the finite field $\mathbb{F}_p$ which realizes either a full threshold access structure or a Q2-access structure. There are various MPC protocols that enable actively secure MPC with abort to be carried out using such an LSSS. For example in the full-threshold case, one can define an LSSS $\langle \cdot \rangle_p$ and an associated MPC protocol, using the SPDZ [DPSZ12] system. For general Q2 (actually *multiplicative*) LSSSs one can utilize the protocol by Smart and Wood [SW19] to produce an MPC protocol directly using the given LSSS. Whereas using a modified LSSS one can do the same using the protocol given by Chida et al. [CGH$^+$18]. The difference between the last two protocols is that [SW19] provides an online phase with lower communication complexity than [CGH$^+$18], at the expense of having to utilize a slightly more expensive offline phase.

We now describe what is meant to share secrets using the LSSS given by $\langle \cdot \rangle_p$, using the well-known description of monotone span programs (MSPs) first given by Karchmer and Widgerson [KW93]. Let $M \in \mathbb{F}_p^{m \times k}$ be a matrix, choose a non-zero "target" vector $\mathbf{t} \in \mathbb{F}_p^k$ and an index function $\iota : \{1, \ldots, m\} \longrightarrow \{1, \ldots, n\}$. To share a secret s using these data, the dealer samples a vector $\mathbf{k} \leftarrow \mathbb{F}_p^k$ such that $\mathbf{t} \cdot \mathbf{k}^\mathsf{T} = s \in \mathbb{F}_p$, sets $\mathbf{s} = (s_1, \ldots, s_m) = M \cdot \mathbf{k}$, and for each $j \in [m]$ computes $i = \iota(j)$ and sends s_i to party P_i over a secure channel. The matrix is chosen in such a way that for any qualified set of parties, there is a (public) recombination vector $\mathbf{r}$ that given the share vector $\mathbf{s}$ (i.e. the concatenation of shares held by the qualified set of parties) can recover the secret by computing $s = \mathbf{r} \cdot \mathbf{s}^\mathsf{T}$.

To ensure correctness in the case of a malicious adversary, we can add authentication to LSSSs in different ways, depending on the access structure. In the SPDZ protocol, every secret has a linear information-theoretic message authentication code (MAC) attached to it: the parties hold a secret-shared global MAC key $\langle \alpha \rangle_p$, and for every secret x, the parties hold $\langle x \rangle_p$ and $\langle \alpha \cdot x \rangle_p$. At the end of a computation, the MACs are checked to reveal whether such additive errors were introduced. This means that in order to change a secret in the scheme (without detection) by adding error ε to a share of the secret, the adversary has to modify a share on the corresponding MAC share by $\alpha \cdot \varepsilon$ – i.e. it must correctly guess the MAC key α. For a Q2 access structure, it has been shown [SW19] that error-detection comes essentially "for free" using a kind of internal redundancy of the LSSS.

Both of these approaches to authentication are linear, so in fact they can be described in essentially the same way, as follows. First, note that the MAC'd SPDZ sharing can be described as above by setting the target vector as $\mathbf{t} = (1, \ldots, 1, 0, \ldots, 0) \in \mathbb{F}_p^{2 \cdot n - 1}$, the index function as $\iota(i) = \iota(n + i) = i$, for $i \in [n]$, and $M = (I_{2n-1} | \mathbf{m}^\mathsf{T})^\mathsf{T}$, where $\mathbf{m} = (\alpha, \ldots, \alpha, -1, \ldots, -1)$. There is a matrix H (the parity-check matrix of the code generated by M) which takes the full set of shares $\mathbf{s}$ (a vector consisting of both shares of the secret and shares of the corresponding MAC) and checks whether they are *valid*. A set of shares is the output of the above procedure if and only if $H \cdot \mathbf{s}^\mathsf{T} = \mathbf{0}$.

In MPC based on Q2 access structures, the error-detection process involves the same multiplication by a parity-check matrix H of the LSSS matrix, where in this case the matrix H is public, whereas for SPDZ the matrix H is secret. The difference between the protocols in the Q2 and full threshold case are (essentially) in how one uses H to check shares are valid, i.e. to detect whether the protocol should abort or not. We will denote by $[\![\cdot]\!]_p$ values in $\mathbb{F}_p$ that are secret shared according to an LSSS and authenticated.

2.2 BMR-style MPC for Binary Circuits

Our multiparty GC-based protocol is a combination, with some extensions, of protocols described in [FKOS15, HSS17, KOS15, WRK17]. We let $\mathbb{K} = \mathbb{F}_{2^\kappa}$ be a finite field, recall we use $\mathbb{K} = \mathbb{F}_{2^{128}}$ in our implementation. For each element $x \in \mathbb{F}_2$ (resp. $\mathbb{K}$), we denote $\langle x \rangle_2$ (resp. $\langle x \rangle_{\mathbb{K}}$) the *unauthenticated* additive sharing of x over $\mathbb{F}_2$ (resp. $\mathbb{K}$), where $x = \sum_{i \in [n]} x_i$, with party P_i holding $x_i \in \mathbb{F}_2$ (resp. $\mathbb{K}$).

Similarly to the $\mathbb{F}_p$ case, we denote by $[\![b]\!]_2$ a bit that is linearly secret shared according to $\langle \cdot \rangle_2$ and *authenticated* according to the pairwise BDOZ-style MAC introduced by Bendlin et al. [BDOZ11]. This means that every party P_i authenticate their share b_i towards party P_j, for each $j \neq i$, by holding a MAC $M_i^j \in \mathbb{K}$, such that

$$M_i^j = K_j^i + b_i \cdot \Delta_j \in \mathbb{K},$$

where P_j holds the local key $K_j^i \in \mathbb{K}$ and the fixed global MAC key $\Delta_j \in \mathbb{K}$. In what follows, we let $\underline{\Delta}$ denote the vector $(\Delta_1, \ldots, \Delta_n)$, and denote by $[\![b_i]\!]_2^{ij}$ this 2-party authenticated sharing of the bit b_i. We can also define an n-party representation of a bit $b = \sum_{i=1}^n b_i$, where each P_i holds the bit-share b_i, $n-1$ MACs M_i^j, $n-1$ local keys K_i^j and Δ_i, i.e.

$$[\![b]\!]_2 = \{b_i, \Delta_i, \{M_i^j, K_i^j\}_{j \neq i}\}_{i \in [n]}.$$

Clearly, two sharings can be added together and multiplied by a constant in the trivial way, so that also the binary $[\![\cdot]\!]_2$-representation is linear. Addition of constants, $[\![a]\!]_2 \leftarrow [\![b]\!]_2 + c$ for $c \in \{0,1\}$, is done by setting

$$a_1 \leftarrow b_1 + c \quad \text{and} \quad a_i \leftarrow b_i \quad \text{for } i \neq 1,$$

with the MAC values and keys being updated by setting

$$\begin{aligned} M_1^j[a] &\leftarrow M_1^j[b], \quad K_j^1[a] = K_j^1[b] \oplus c \cdot \Delta_j \quad \text{for } 1 \leq j \leq n, \\ M_i^j[a] &\leftarrow M_i^j[b], \quad K_j^i[a] \leftarrow K_j^i[b] \quad \text{for } i \neq 1, 1 \leq j \leq n. \end{aligned}$$

2.3 Zaphod - Online Functionality

Our MPC protocol is defined in the so-called pre-processing model, with an offline phase that provides all the correlated randomness needed for the online computation. In this section we explain the online functionality we aim to provide. Our goal is to describe a reactive MPC protocol which combines *both* arithmetic and boolean circuits. When using the mapping from binary to arithmetic circuits we will assume the data being transferred has bit length bounded by ℓ. In other words every data item x can be represented using in ℓ bits or is equivalently an element of $\mathbb{Z}_{\langle 2^\ell \rangle} = \{-2^{\ell-1}+1, \ldots, 2^{\ell-1}\}$. In our implementation we select

$\ell = 64$, but of course in practice ℓ can vary depending on the application. We select $\ell = 64$ so as to easily map onto the *unsigned int* data-type known to programmers.

As mentioned before, we let sec denote a security parameter related to the statistical security of our protocols. To enable more efficient protocols we will make one of the following three assumptions.

1. Either $\ell + \mathsf{sec} < \log_2 p$,
2. Or $(p - 2^{\lfloor \log_2 p \rfloor})/p < 2^{-\mathsf{sec}}$,
3. Or $(2^{\lceil \log_2 p \rceil} - p)/2^{\lceil \log_2 p \rceil} < 2^{-\mathsf{sec}}$.

In each of these cases we guarantee that selecting $\log_2 p$ bits b_i at random and then forming, for $x \in \mathbb{Z}_{\langle 2^\ell \rangle}$,

$$x + \sum_{i=0}^{\lceil \log_2 p \rceil} b_i \cdot 2^i \pmod{p}$$

will statistically hide the value of x. The assumptions are needed as the value $\sum b_i \cdot 2^i \pmod{p}$ is not necessarily a uniformly random element from $\mathbb{F}_p$.

Functionality $\mathcal{F}_{\mathsf{MPC}}$ - Zaphod Evaluation

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 $(\mathit{varid}, \mathit{domain}, x)$, where $\mathit{varid} \in I$, $\mathit{domain} \in \{\mathbb{F}_2^\ell, \mathbb{F}_p\}$ and $x \in \mathit{domain}$. We assume p is restricted as in the main text.

Initialize: On input (Init) from all parties, the functionality activates.
If (Init) was received before, do nothing.
Input: On input $(\mathit{Input}, P_i, \mathit{varid}, \mathit{domain}, x)$ from P_i and $(\mathit{input}, P_i, \mathit{varid}, \mathit{domain})$ from all other parties, with varid a fresh identifier, store $(\mathit{varid}, \mathit{domain}, x)$.
Evaluate: Upon receiving $(\{\mathit{varid}_j\}_{j \in m}, \mathit{varid}, \mathit{domain}, C_{\bar{f}})$, from all parties, where $\bar{f} : \{\mathit{domain}\}^m \to \mathit{domain}$ and varid is a fresh identifier, if $\{\mathit{varid}_j\}_{j \in [m]}$ were previously stored, proceed as follows:
1. Retrieve $(\mathit{varid}_j, \mathit{domain}, x_j)$, for each $j \in [m]$
2. Store $(\mathit{varid}, \mathit{domain}, x_{m+1} \leftarrow \bar{f}(x_1, \ldots, x_m))$

Output: On input $(\mathit{Output}, \mathit{varid}, \mathit{domain}, \mathit{type})$, from all parties (if varid is present in memory):
1. If $\mathit{type} = 0$ (**Public Output**): Retrieve (varid, y) and send it to $\mathcal{A}$. If the adversary sends $\mathsf{Deliver}$, send y to all parties.
2. If $\mathit{type} = -1$ (**No Output**): Retrieve (varid, y) and send it to $\mathcal{A}$. If the adversary sends $\mathsf{Deliver}$, store (varid, y) and continue .
3. Otherwise (**Private Output**): Send (varid) to $\mathcal{A}$. Upon receiving $\mathsf{Deliver}$ from $\mathcal{A}$, send y to P_i

Abort: The adversary can at any time send *abort*, upon which send *abort* to all honest parties and halt.

Figure 2. The ideal functionality for MPC with Abort over $\mathbb{F}_p$ and $\mathbb{F}_2^\ell$ - Evaluation

In Figure 2 and Figure 3 we provide the functionality for our MPC black box. Each value in $\mathcal{F}_{\mathsf{MPC}}$ is uniquely identified by an identifier $\mathit{varid} \in I$, where I is a set of valid identifiers, and a domain set $\mathit{domain} \in \{\mathbb{F}_p, \mathbb{F}_2^\ell\}$. We model the functionality in such a way that it is independent of the details of the authentication technique used in the protocol, and captures all the MPC computations enabled by the `SCALE-MAMBA` system. This abstraction, when instantiated with $\mathbb{F}_p$, captures both the SPDZ protocol and the Q2 protocol from [SW19], but not the protocol from [CGH+18]. The modifications needed to also encompass the protocol from [CGH+18] are straightforward, but would lead to a more complex description from which we spare the reader. When the functionality is instantiated with $\mathbb{F}_2^\ell$, it is intended to implement the BMR-protocol. Also note that our abstract MPC functionality provides actively secure MPC with abort.

Hence, one can see $\mathcal{F}_{\mathsf{MPC}}$- **Zaphod Evaluation** as a combination of two MPC black boxes, specified by the set assigned to *domain*, along with two conversion routines, namely *ConvertToField* and *ConvertToBinary*, given in $\mathcal{F}_{\mathsf{MPC}}$- **Zaphod Conversion**. If *domain* $= \mathbb{F}_p$, the MPC black box provides arithmetic operations over the finite field $\mathbb{F}_p$, whereas if *domain* $= \mathbb{F}_2^\ell$, it enables one to execute arbitrary binary circuits over binary vectors of length ℓ, i.e. functions with arguments and results in the set $\mathbb{F}_2^\ell$. Note that the execution of **Output** with *type* $= -1$ is a command required for simulation in reactive functionalities.

$\mathcal{F}_{\mathsf{MPC}}$- **Zaphod Conversion** permits parties to switch between the two MPC black boxes. Note that we have defined *ConvertToBinary* and *ConvertToField* to ensure that they are mutual inverses of each other (if the $\mathbb{F}_p$-input element is fewer than ℓ bits in length when in the centred interval $(-p/2, \dots, p/2]$).

Functionality $\mathcal{F}_{\mathsf{MPC}}$ - Zaphod Conversion

Convert To Field: On input $(\mathit{Convert}, \mathit{varid}_1, \mathbb{F}_2^\ell, \mathit{varid}_2, \mathbb{F}_p)$:
1. Retrieve $(\mathit{varid}_1, \mathbb{F}_2^\ell, \mathbf{x})$ and convert $\mathbf{x}$ to an element $y \in \mathbb{F}_p$ by setting $y \leftarrow -x_{\ell-1} \cdot 2^{\ell-1} + \sum_{i=0}^{\ell-2} x_i \cdot 2^i$.
2. Store $(\mathit{varid}_2, \mathbb{F}_p, y)$.

Convert To Binary On input $(\mathit{Convert}, \mathit{varid}_1, \mathbb{F}_p, \mathit{varid}_2, \mathbb{F}_2^\ell)$:
1. Retrieve $(\mathit{varid}_1, \mathbb{F}_p, x)$ as an integer in the range $(-p/2, \dots, p/2)$.
2. Express $y = x \pmod{2^\ell}$ as $y = \sum_{i=0}^{\ell} y_i \cdot 2^i$ for $y_i \in \{0, 1\}$
3. Consider the values y_i as elements in $\mathbb{F}_2$ and pack them into a vector $\mathbf{y} \in \mathbb{F}_2^\ell$.

Figure 3. Extending the ideal functionality for MPC over $\mathbb{F}_p$ and $\mathbb{F}_2^\ell$ to allow conversion

3 Pre-processing

Here we describe the entire pre-processing phase of Zaphod. At a high level, we need to produce all the correlated randomness needed to implement both the arithmetic and binary MPC functionalities described in the previous section, and the conversion between them.

3.1 SPDZ and TinyOT-style Pre-processing

We recall here that the main task of the offline phase in our LSSS-based MPC in the pre-processing model with active security is to produce random authenticated secret-shared values and random authenticated multiplication triples to allow the input and the multiplication steps during the online evaluation.

This is formalized by the ideal functionality $\mathcal{F}_{\mathsf{Prep}}[\mathsf{AuthLSSS}]$ described in Figure 4. It is a general pre-processing functionality for LSSS-based MPC, that allows parties to choose between arithmetic and binary fields using the **Initialize** command.

Again, we model our functionality so that it is independent of the authentication method used in the protocol that implements it. However, when the domain set is fixed we concretely instantiate the handle on values with $[\![\cdot]\!]_p$ if $domain = \mathbb{F}_p$, and with $[\![\cdot]\!]_2$ otherwise. We denote the $\mathbb{F}_p$-instantiations by $\mathcal{F}_{\mathsf{Prep}}[\mathsf{SPDZ}]$.

After the initialization, $\mathcal{F}_{\mathsf{Prep}}[\mathsf{AuthLSSS}]$ allows parties to provide private inputs (**Input**), perform linear operations (**LinearComb**), produce random secret-shared bits (**Random-Bit**) and triples (**Triple**), and reveal values either publicly to every party (**Open**) or privately to a single party (**OpenTo**). These open commands permit the adversary to output inconsistent values or shares, so to ensure correctness at a later point in time, the functionality provides the **Check** command. Since we aim for security with abort, the adversary can at any point send the message *abort* to the functionality (**Abort**).

This general pre-processing functionality can be implemented in different ways [DPSZ12, KOS16, NNOB12, KPR18, SW19] depending on the access structure. In particular, in the `SCALE-MAMBA` system (when $\mathbb{F} = \mathbb{F}_p$) the functionality $\mathcal{F}_{\mathsf{Prep}}[\mathsf{SPDZ}]$ is implemented according to the Overdrive-High Gear variant of Keller et al. [KPR18], in the full threshold case, and by the protocol in [SW19], in the Q2 case. We call both these instantiations $\Pi_{\mathsf{Prep}}[\mathsf{SPDZ}]$. We refer the reader to these papers for a complete description of the protocols and the corresponding security proofs.

Functionality $\mathcal{F}_{\mathsf{Prep}}[\mathsf{AuthLSSS}]$

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 identifiers, all values are stored in the form $(varid, domain, x)$, where $varid \in I$ and $domain = \{\mathbb{F}_2, \mathbb{F}_p\}$.

Initialize: On input $(Init, \mathbb{F})$ from all parties, the functionality stores $(domain, \mathbb{F})$, where $\mathbb{F}$ is either $\mathbb{F}_2$ or $\mathbb{F}_p$.
Input: On input $(Input, P_i, varid, domain, x)$ from party P_i and $(input, P_i, varid)$ from all other parties, with $varid$ a fresh identifier and $x \in domain$, store $(varid, x)$.
RandomBit: On input $(RandomBit, domain, varid)$ from all parties, with $varid$ a fresh identifier, sample $b \in \{0, 1\}$, store $(varid, b)$.
LinearComb: On input $(LinearComb, domain, varid_1, varid_2, varid_3,\ a,\ b,\ c)$ from all parties, where $a, b, c \in domain$, retrieve $(varid_1, x)$, $domain$,$(varid_2, y)$ and store $(varid_3, domain, a \cdot x + b \cdot y + c)$.
Triple: On command $(Triple, domain, varid_1, varid_2, varid_3)$, for fresh identifiers $varid_1, varid_2, varid_3$, select $a, b \in domain$ at random and set $c \leftarrow a \cdot b$. Store $(varid_1, domain, a)$, $(varid_2, domain, b)$, $(varid_3, domain, c)$.
Open: On input $(Open, domain, varid)$ from all parties (if $varid$ is present in memory), functionality retrieve $(varid, y)$ and output y to the adversary. Wait for an input y' from $\mathcal{A}$. If this input is $\mathsf{Deliver}$ then output y' to all parties. Otherwise send *abort* to honest parties and halt.
OpenTo: On input $(Open, domain, varid, i)$ from all parties (if $varid$ is present in memory), retrieve $(varid, y)$. If $i \in A$, send y to $\mathcal{A}$ and wait for a reply, otherwise just wait for an input from $\mathcal{A}$. Output according to the adversary response.
Check: On input $(Check, varid_1, \ldots, varid_t, x_1, \ldots, x_t)$ from all parties, wait for an input from the adversary. If the adversary inputs *Proceed*, store the value x_i, for all $i \in [t]$, and output *Proceed* to all parties, otherwise output *abort* and halt.
Abort: The adversary can at any time send *abort*, upon which send *abort* to all honest parties and halt.

Figure 4. The ideal preprocessing functionality for MPC over $\mathbb{F}_p$ and $\mathbb{F}_2$. We denote this functionality by $\mathcal{F}_{\mathsf{Prep}}[\mathsf{SPDZ}]$ when $domain = \mathbb{F}_p$.

3.2 BMR Preprocessing

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

This functionality runs with two parties P_S (the sender) and P_R (the receiver) and an ideal adversary $\mathcal{A}$.

Initialize: On input (Init, Δ), $\Delta \in \mathbb{K}$ from P_S and (Init) from P_R, store Δ.
Extend: On input $(\mathit{Extend}, b_1, \ldots, b_m)$ from P_R and (Extend) from P_S proceed as follows:
1. Sample $t_i \in \mathbb{K}$, for $i \in [m]$. If P_R is corrupted receive t_i from $\mathcal{A}$.
2. Set $q_i \leftarrow t_i + b_i \cdot \Delta \in \mathbb{K}$, for any $i \in [m]$. If P_s is corrupt, receive q_i from $\mathcal{A}$ and set $t_i \leftarrow q_i + b_i \cdot \Delta$ instead.
3. Output t_i to P_R and q_i to P_S, for $i \in [m]$

Figure 5. The ideal functionality for COT

In the pre-processing phase of modern variants of the BMR protocol, all parties contribute to the garbling of each gate individually. We now describe our implementation of the BMR pre-processing protocol $\Pi_{\mathsf{Prep}}[\mathtt{BMR}]$ which is fully described in Figure 6.

$\mathcal{F}_{\mathsf{n-TinyOT}}$ functionality The protocol $\Pi_{\mathsf{Prep}}[\mathtt{BMR}]$ uses the ideal functionality $\mathcal{F}_{\mathsf{Prep}}[\mathsf{AuthLSSS}]$ described in the previous section with $\mathbb{F} = \mathbb{F}_2$. However, to achieve concrete efficiency and ease the description, we consider a slightly modified version of $\mathcal{F}_{\mathsf{Prep}}[\mathsf{AuthLSSS}]$ that is better suited for the $[\![\cdot]\!]_2^{ij}$ and $[\![\cdot]\!]_2$ representations described in Section 2.2, similarly to what was done by Hazay et al. [HSS17]. We call this modified functionality $\mathcal{F}_{\mathsf{n-TinyOT}}$. (See [HSS17] for a complete description of it.) We summarize here the main differences compared to $\mathcal{F}_{\mathsf{Prep}}[\mathsf{AuthLSSS}]$:

- $\mathcal{F}_{\mathsf{n-TinyOT}}.\mathsf{Initialize}$: The functionality receives from the honest parties and the adversary the MAC key shares $\Delta_i \in \mathbb{K}$, for $i \in [n]$.
- $\mathcal{F}_{\mathsf{n-TinyOT}}.\mathsf{Get}$: On input $(\mathsf{Get}, \mathbb{F}_2, \mathit{varid}, i)$ from all parties, the functionality samples a random $b_i \in \mathbb{F}_2$ if $i \notin A$, otherwise receives b_i from the adversary. Store $(\mathit{varid}, \mathbb{F}_2, b_i)$. This command is used by $\mathcal{F}_{\mathsf{n-TinyOT}}$ instead of the private input command in $\mathcal{F}_{\mathsf{Prep}}[\mathsf{AuthLSSS}]$.
- $\mathcal{F}_{\mathsf{n-TinyOT}}.\mathsf{Triple}$: On input $(\mathsf{Triple}, \mathbb{F}_2, \{\mathit{varid}_i\}_{i\in[3n+3]})$ the functionality selects $a, b \in \mathbb{F}_2$ at random and set $c \leftarrow a \cdot b$. It receives shares a_i, b_i, c_i from $\mathcal{A}$, for $i \in A$, and samples a_i, b_i, c_i, $i \notin A$, subject to $\sum_i a_i = a, \sum_i b_i = b, \sum_i c_i = c$. It stores these values with the corresponding identifiers.

All the other commands, as $\mathcal{F}_{\mathsf{n-TinyOT}}.\mathsf{Open}$ and $\mathcal{F}_{\mathsf{n-TinyOT}}.\mathsf{OpenTo}$, are similar to those in $\mathcal{F}_{\mathsf{Prep}}[\mathsf{AuthLSSS}]$.

Essentially, this functionality enables the parties to produce random authenticated bits and bit-triples to be used in the protocol $\Pi_{\mathsf{Prep}}[\mathtt{BMR}]$. To implement $\mathcal{F}_{\mathsf{n-TinyOT}}$ in the $\mathcal{F}_{\mathsf{COT}}$-hybrid model (where the functionality $\mathcal{F}_{\mathsf{COT}}$ is described in Figure 5), we use a combination of protocols from [KOS15, FKOS15, HSS17, WRK17] as follows. We implement $\mathcal{F}_{\mathsf{COT}}$ using the OT-extension protocols described in the two works [FKOS15, Full Version, Figure 19] and [KOS15, Full Version, Figure 7]. These correlated OTs are easily converted into random sharings of authenticated bits (called aShares or aBits in previous papers and $[\![\cdot]\!]_2$-sharings in this paper), by setting $M_i^j = t_i$ and $K_j^i = q_i$, where t_i and q_i are the output of $\mathcal{F}_{\mathsf{COT}}$. For this step we use [HSS17, Full Version, Figure 16]. Finally, these authenticated bits are converted into triples (aANDs in other works) using [WRK17, Full Version, Figures 16, 8 and 18 in order].

Garbling Using $\mathcal{F}_{\mathsf{n-TinyOT}}$ we can now describe the protocol $\Pi_{\mathsf{Prep}}[\mathtt{BMR}]$. It is a (*minor*) modification of the n-party garbled circuit construction given in [HSS17]. The main difference is that we are interested in a reactive functionality, i.e. we are going to evaluate multiple different functions F, with the inputs and outputs of this function being vectors of authenticated bits $[\![\mathbf{x}]\!]_2$. The function we want to compute will be given by

$$[\![\mathbf{y}]\!]_2 = F([\![\mathbf{x}]\!]_2),$$

where $[\![\mathbf{x}]\!]_2$ and $[\![\mathbf{y}]\!]_2$ are vectors of authenticated bits. The number of bits in a vector is denoted by $|\mathbf{x}|$, and the j-th bit of $\mathbf{x}$ is denoted by $\mathbf{x}^j$. The common input of all parties in $\mathcal{P}$ is the function to be evaluated given as a binary *combinatorial* circuit consisting of three types of gates: XOR, AND and INV. The circuit representing F is assumed to be given in a topological order.

We describe at a high level the precise garbling being used, and why it works. First, the parties call the $\mathcal{F}_{\mathsf{n-TinyOT}}$ functionality on command **Initialize** to obtain Δ_i, for $i \in [n]$. To each wire w in the circuit C_F we will associate a random wire mask $[\![\lambda_w]\!]_2$ that the parties obtain by calling $\mathcal{F}_{\mathsf{n-TinyOT}}.\mathsf{Get}$. It will be used to mask the actual value v_w of the wire w, so to define the *external value* as $\epsilon_w = v_w \oplus \lambda_w$. There is a special wire, called the *one wire*, which is used to represent the value one (needed in the INV gate which is given by $\mathsf{INV}(a) = \mathsf{XOR}(a, 1)$). This always has wire mask $\lambda_{\mathbf{one}} = 0$ and, consequently, external value $\epsilon_{\mathbf{one}} = 1$.

The wire masks are chosen independently for all wires in the circuit, *except* for the outputs of XOR gates, where if we have a gate $c = \mathsf{XOR}(a, b)$ then we define $\lambda_c = \lambda_a \oplus \lambda_b$. Note, since an INV gate is a XOR gate this also applies to the outputs of INV gates where we write for $b = \mathsf{INV}(a)$, $\lambda_b = \lambda_a \oplus 1$.

Associated to each wire w, we also have a *vector* of secret *base* keys, $\mathbf{k}_{w,0} = (\mathbf{k}^1_{w,0}, \ldots, \mathbf{k}^n_{w,0}) \in \mathbb{K}^n$. These are the garbled circuit keys associated with the zero value on this wire. To enable free-XOR, the value for the one wire is given by $\mathbf{k}_{a,1} = \mathbf{k}_{a,0} + \underline{\Delta}$, where we $\underline{\Delta} = (\Delta_1, \ldots, \Delta_n)$ is the *global difference* vector. The key observation [HSS17] is that each party P_i uses the same Δ_i as in the $[\![\cdot]\!]_2$-representation, i.e. the MAC keys used to authenticate in $\mathbb{F}_2$ and the global differences used in BMR are the same. The wire keys are often called wire labels in the literature. The *one wire* also has a set of base keys given by a vector $\mathbf{k}_{\mathbf{one},0}$.

We will require a PRF, denoted by $F_{k_0,k_1}(m)$, which takes as input a key in $\mathbb{K}^2$, a one block message m and outputs an element in $\mathbb{K}$. The precise PRF we use, since we select $\mathbb{K} = \mathbb{F}_{2^{128}}$ is derived from Matyas–Meyer–Oseas hashing

$$M = m \oplus (k_0 \cdot X) \oplus (k_1 \cdot X^2),$$

$\Pi_{\mathsf{Prep}}[\mathtt{BMR}]$

COMMON INPUT: All the parties hold the circuit C_F to be evaluated in a topological order.

Initialize: Each party P_i calls $\mathcal{F}_{\mathsf{n-TinyOT}}$ to obtain the global difference $\Delta_i \in \mathbb{K}$.
Garbling:
- For all wires w which are either an input wire to the circuit or the output wire of an AND gate compute the shared wire mask $[\![\lambda_w]\!]_2 \leftarrow \mathcal{F}_{\mathsf{n-TinyOT}}.\mathsf{Get}()$.
- For all wires w which are either an input wire or the output wire of an AND gate party P_i samples $\mathbf{k}^i_{w,0} \in \mathbb{K}$ at random and sets $\mathbf{k}^i_{w,1} \leftarrow \mathbf{k}^i_{w,0} + \Delta_i$.
- Each P_i selects $\mathbf{k}^i_{\mathbf{one},0} \in \mathbb{K}$ at random and sets $\mathbf{k}^i_{\mathbf{one},1} \leftarrow \mathbf{k}^i_{\mathbf{one},0} \oplus \Delta_i$.
- The parties now progress through the circuit in the given topological ordering and process the gates as follows:
  1. $c = \mathsf{XOR}(a, b)$ Gate:
     - Each party sets $[\![\lambda_c]\!]_2 \leftarrow [\![\lambda_a]\!]_2 \oplus [\![\lambda_b]\!]_2$, $\mathbf{k}^i_{c,0} \leftarrow \mathbf{k}^i_{a,0} \oplus \mathbf{k}^i_{b,0}$ and $\mathbf{k}^i_{c,1} \leftarrow \mathbf{k}^i_{a,0} \oplus \Delta_i$.
  2. $c = \mathsf{INV}(a)$ Gate:
     - Each party sets $[\![\lambda_c]\!]_2 \leftarrow [\![\lambda_a]\!]_2 \oplus 1$, $\mathbf{k}^i_{c,0} \leftarrow \mathbf{k}^i_{a,0} \oplus \mathbf{k}^i_{\mathbf{one},0}$ and $\mathbf{k}^i_{c,1} \leftarrow \mathbf{k}^i_{a,0} \oplus \Delta_i$.
- For each AND gate $c = \mathsf{AND}(a, b)$ the parties now execute:
  - The parties runs the subprotocol Π_{Mult} to compute $[\![a]\!]_2 \cdot [\![b]\!]_2$ and obtain $[\![\lambda_{ab}]\!]_2 \leftarrow \Pi_{\mathsf{Mult}}([\![\lambda_a]\!]_2, [\![\lambda_b]\!]_2)$.
  - Each party locally obtains a share of $\langle \lambda_a \cdot \Delta_j \rangle_{\mathbb{K}}$, $\langle \lambda_b \cdot \Delta_j \rangle_{\mathbb{K}}$ and $\langle (\lambda_{ab} + \lambda_c) \cdot \Delta_j \rangle_{\mathbb{K}}$.
  - For each $\epsilon_a, \epsilon_b \in \{0, 1\}$ and $j \in [n]$ party i computes its share of $\langle \rho_{j,\epsilon_a,\epsilon_b} \rangle_{\mathbb{K}} = \langle (\lambda_{ab} + \lambda_c) \cdot \Delta_j \rangle_{\mathbb{K}} \oplus (\epsilon_a \cdot \langle \lambda_b \cdot \Delta_j \rangle_{\mathbb{K}}) \oplus (\epsilon_b \cdot \langle \lambda_a \cdot \Delta_j \rangle_{\mathbb{K}}) \oplus (\epsilon_a \cdot \epsilon_b \cdot \Delta_j)$.
  - Party i can now compute its share of the j column of the garbled gate via

$$\langle g_{j,\epsilon_a,\epsilon_b} \rangle^j_{\mathbb{K}} \leftarrow \langle \rho_{j,\epsilon_a,\epsilon_b} \rangle^j_{\mathbb{K}} \oplus F_{\mathbf{k}^j_{a,\epsilon_a}, \mathbf{k}^j_{b,\epsilon_b}}(g \| j) \oplus \mathbf{k}^j_{c,0}$$

    if $j = i$, and otherwise via

$$\langle g_{j,\epsilon_a,\epsilon_b} \rangle^i_{\mathbb{K}} \leftarrow \langle \rho_{j,\epsilon_a,\epsilon_b} \rangle^i_{\mathbb{K}} \oplus F_{\mathbf{k}^i_{a,\epsilon_a}, \mathbf{k}^i_{b,\epsilon_b}}(g \| j),$$

    where g is a unique identifier for this AND-gate.

Open Garbling: The parties open the garbled gates $\langle g_{j,\epsilon_a,\epsilon_b} \rangle_{\mathbb{K}}$, and sum the shares so as to obtain, for each $\epsilon_a, \epsilon_b \in \{0, 1\}$ and $j \in [n]$,

$$g_{j,\epsilon_a,\epsilon_b} \leftarrow \mathbf{k}^i_{c,\epsilon_c} \oplus \Big(\bigoplus_{j=1}^{n} F_{\mathbf{k}^j_{a,\epsilon_a}, \mathbf{k}^j_{b,\epsilon_b}}(g \| j) \Big).$$

The party i sends $\mathbf{k}^i_{\mathbf{one},1}$ to all parties.

Figure 6. The BMR pre-prcessing protocol

$$F_{k_0,k_1}(m) = \sigma(M) \oplus \mathsf{AES}_{\mathbf{0}}(\sigma(M)),$$

where $k \cdot X$ denotes multiplication by X in the finite field $\mathbb{K}$ (for X the formal root of the defining polynomial of $\mathbb{K}$, i.e. we have $\mathbb{K} = \mathbb{F}_2/(f(X))$, and $\sigma(M)$ is the function on 128-bit datablocks defined in [GKWY19] by $\sigma(M_L \| M_R) = (M_R \oplus M_L) \| M_L$, where M_L and M_R are the left and right halves of the message M respectively.

A *garbled gate* is a means of translating the input wire keys (for actual wire values of 0 or 1) into the wire keys for the output of the gate. These are only needed for AND gates (and any other non-linear gate one wishes to support). The garbled gate, associated with $c = \mathsf{AND}(a, b)$, is indexed by the *external values* ϵ_a and ϵ_b of the input wires. Note, in this case we have $v_c = v_a \cdot v_b$ as we are evaluating an AND gate. The key equation is the following one:

$$\begin{aligned} E(\lambda_a, \lambda_b, \lambda_c, \epsilon_a, \epsilon_b) &= ((\lambda_a \cdot \lambda_b) \oplus \lambda_c) \oplus (\epsilon_a \cdot \lambda_b) \\ &\quad \oplus (\epsilon_b \cdot \lambda_a) \oplus (\epsilon_a \cdot \epsilon_b) \\ &= ((\lambda_a \cdot \lambda_b) \oplus \lambda_c) \oplus ((\lambda_b \cdot v_a) \oplus (\lambda_b \cdot \lambda_a)) \\ &\quad \oplus ((\lambda_a \cdot v_b) \oplus (\lambda_a \cdot \lambda_b)) \\ &\quad \oplus ((\lambda_b \cdot v_a) \oplus (v_a \cdot v_b)) \\ &\quad \oplus ((\lambda_a \cdot v_b) \oplus (\lambda_a \cdot \lambda_b)) \\ &= (\lambda_c \oplus v_c) \\ &= \epsilon_c. \end{aligned}$$

The key trick in [HSS17] is to enable the Tiny-OT authenticated shares $[\![\lambda_a]\!]_2$, $[\![\lambda_b]\!]_2$ and $[\![\lambda_c]\!]_2$ to be converted *without interaction* into a sharing $\langle \rho_{\epsilon_a,\epsilon_b,j} \rangle_{\mathbb{K}} = \langle E(\lambda_a, \lambda_b, \lambda_c, \epsilon_a, \epsilon_b) \cdot \Delta_j \rangle_{\mathbb{K}}$ for every value of $j \in [n]$. Thus each party P_i can derive a share $\langle \rho_{\epsilon_a,\epsilon_b,j} \rangle^i_{\mathbb{K}}$, for the values $\epsilon_a, \epsilon_b \in \{0, 1\}$ without knowing the values of λ_a, λ_b and λ_c. This is done via the following Lemma

Lemma 3.1. *Let $[\![x]\!]_2$ be an authenticated bit with value/share/MAC values (x_i, M_i^j, K_i^j) then*

$$\begin{aligned} y_i &= M_i^j \text{ for } i \neq j, \\ y_j &= (x_j \cdot \Delta_j) \oplus \Big(\bigoplus_{k \neq j} K_k^j \Big). \end{aligned}$$

is a $\langle \cdot \rangle_{\mathbb{K}}$-sharing of $x \cdot \Delta_j$.

Proof.

$$\begin{aligned} \bigoplus_{i=1}^{n} y_i &= (x_j \cdot \Delta_j) \oplus \Big(\bigoplus_{k \neq j} K_k^j \Big) \oplus \Big(\bigoplus_{i \neq j} M_i^j \Big) = (x_j \cdot \Delta_j) \oplus \Big(\bigoplus_{k \neq j} (K_k^j \oplus M_k^j) \Big) \\ &= (x_j \cdot \Delta_j) \oplus \Big(\bigoplus_{k \neq j} (x_k \cdot \Delta_j) \Big) = \bigoplus_{i=1}^{n} (x_i \cdot \Delta_j) \end{aligned}$$

$$= x \cdot \Delta_j.$$

□

To encrypt the gate each party P_i encrypts their *output* base wire key $\mathbf{k}^i_{c,0}$ using the PRF applied to the external wire keys $\mathbf{k}^i_{a,\epsilon_a}$ and $\mathbf{k}^i_{b,\epsilon_b}$. They also add on the sharing $\langle \rho_{\epsilon_a,\epsilon_b,j} \rangle^i_{\mathbb{K}}$. The other parties for this output key (column) do a similar operation but do not add on the base key. Thus in column i (corresponding to the garbling of the ith players base key) for row $(\epsilon_a, \epsilon_b) = (1, 0)$ (say) of the garbled table for gate g we have the following n entries:

$$\begin{gathered}
\langle \rho_{0,1,i} \rangle^{(1)}_{\mathbb{K}} \oplus F_{\mathbf{k}^{(1)}_{a,0}, \mathbf{k}^{(1)}_{b,1}}(g \| 1) \\
\vdots \\
\langle \rho_{0,1,i} \rangle^{(i-1)}_{\mathbb{K}} \oplus F_{\mathbf{k}^{(i-1)}_{a,0}, \mathbf{k}^{(i-1)}_{b,1}}(g \| (i-1)) \\
\langle \rho_{0,1,i} \rangle^{(i)}_{\mathbb{K}} \oplus F_{\mathbf{k}^{(i)}_{a,0}, \mathbf{k}^{(i)}_{b,1}}(g \| i) \oplus \mathbf{k}^{(i)}_{c,0} \\
\langle \rho_{0,1,i} \rangle^{(i+1)}_{\mathbb{K}} \oplus F_{\mathbf{k}^{(i+1)}_{a,0}, \mathbf{k}^{(i+1)}_{b,1}}(g \| (i+1)) \\
\vdots \\
\langle \rho_{0,1,i} \rangle^{(n)}_{\mathbb{K}} \oplus F_{\mathbf{k}^{(n)}_{a,0}, \mathbf{k}^{(n)}_{b,1}}(g \| n).
\end{gathered}$$

Each garbled gate consists of a table of $(4 \times n)$ elements in K held by party P_i. When one opens the gates at the end of garbling these tables are exchanged and added together.

The sub-protocol Π_{Mult}

$\Pi_{\mathsf{Mult}}([\![x]\!]_2, [\![y]\!]_2)$**:** This produces a sharing of $x \cdot y$.

1. Call *Triple* on $\mathcal{F}_{\mathsf{n-TinyOT}}$ to obtain a triple $([\![a]\!]_2, [\![b]\!]_2, [\![c]\!]_2)$.
2. The players compute $[\![d]\!]_2 \leftarrow [\![a]\!]_2 \oplus [\![b]\!]_2$ and $[\![e]\!]_2 \leftarrow [\![b]\!]_2 \oplus [\![y]\!]_2$.
3. The parties call $\mathcal{F}_{\mathsf{n-TinyOT}}.\mathsf{Open}([\![d]\!]_2)$ and $\mathcal{F}_{\mathsf{n-TinyOT}}.\mathsf{Open}([\![e]\!]_2)$ to obtain d and e.
4. The parties compute $[\![z]\!]_2 \leftarrow [\![c]\!]_2 \oplus d \cdot [\![e]\!]_2 \oplus e \cdot [\![a]\!]_2 \oplus d \cdot e$.
5. The parties output $[\![z]\!]_2$.

Figure 7. Subprotocols on authenticated bits

So at the end of garbling *all* parties how the following value in row $(0, 1)$ and column i of the table

$$\begin{aligned}
&\rho_{0,1,i} \oplus \mathbf{k}^{(i)}_{c,0} \oplus \Big(\bigoplus_{j=1}^{n} F_{\mathbf{k}^{(j)}_{a,0}, \mathbf{k}^{(j)}_{b,1}}(g \| j) \Big) \\
&\qquad = (E(\lambda_a, \lambda_b, \lambda_c, 0, 1) \cdot \Delta_j) \oplus \mathbf{k}^{(i)}_{c,0} \oplus \Big(\bigoplus_{j=1}^{n} F_{\mathbf{k}^{(j)}_{a,0}, \mathbf{k}^{(j)}_{b,1}}(g \| j) \Big)
\end{aligned}$$

$$= (\epsilon_c \cdot \Delta_j) \oplus \mathbf{k}_{c,0}^{(i)} \oplus \Big(\bigoplus_{j=1}^{n} F_{\mathbf{k}_{a,0}^{(j)}, \mathbf{k}_{b,1}^{(j)}}(g \| j) \Big)$$

$$= \mathbf{k}_{c,\epsilon_c}^{(i)} \oplus \Big(\bigoplus_{j=1}^{n} F_{\mathbf{k}_{a,0}^{(j)}, \mathbf{k}_{b,1}^{(j)}}(g \| j) \Big).$$

Since we are interested in reactive computations, in the opening garbling phase we do not reveal the wire mask λ_w associated to the circuit-output wires w, as happens in previous works. Our protocol $\Pi_{\mathsf{Prep}}[\mathtt{BMR}]$ securely implements an ideal functionality $\mathcal{F}_{\mathsf{Prep}}[\mathsf{BMR}]$ with active security in the $\mathcal{F}_{\mathsf{n-TinyOT}}$-hybrid model. We do not give here the details of $\mathcal{F}_{\mathsf{Prep}}[\mathsf{BMR}]$ since this functionality is essentially the same as the one given in [HSS17]. Active security is obtained by using the actively secure $\mathcal{F}_{\mathsf{n-TinyOT}}$ functionality, and the Open command of $\mathcal{F}_{\mathsf{n-TinyOT}}$, which enables us to check the opened shares bits. The modifications of the proofs in [HSS17] to the reactive case are minor, and thus we do not give them.

3.3 Complete Pre-processing Functionality

Other than the correlated randomness needed for the circuit evaluation, both in the LSSS and GC fashion, we need random doubly-authenticated bits (daBits) to perform the conversions in the online protocol. The command **daShare**, together with commands in $\mathcal{F}_{\mathsf{Prep}}[\mathsf{SPDZ}]$ and $\mathcal{F}_{\mathsf{Prep}}[\mathsf{BMR}]$, completes our ideal pre-processing functionality $\mathcal{F}_{\mathsf{Prep}}$, given in Figure 8. Extending functionalities in the UC framework by combining and adding new commands is a well-known technique in the literature, e.g. in [KOS16, RW19].

In the next section we give a new procedure for generating daBits and show that it can be used to realize $\mathcal{F}_{\mathsf{Prep}}$ UC securely, essentially by the same proof as in [RW19].

$\mathcal{F}_{\mathsf{Prep}}$

This functionality has all the commands of $\mathcal{F}_{\mathsf{Prep}}[\mathsf{SPDZ}]$ and all the commands of $\mathcal{F}_{\mathsf{Prep}}[\mathsf{BMR}]$ and additionally has the command **daShare** defined below.

daShare: On input $(\mathit{daShare}, \mathbb{F}_2, \mathit{varid}, \mathit{varid}_0, \ldots, \mathit{varid}_{\ell-1},\ b_1, \ldots, b_\ell)$, where $b_i \in \{0,1\}, i \in [\ell]$, compute $r = \sum_{i=0}^{\ell} b_i \cdot 2^i \in \mathbb{F}_p$ and store $(\mathit{varid}, \mathbb{F}_p, r)$.

Figure 8. Ideal functionality for Preprocessing for Zaphod

4 daBits

It was shown by Rotaru and Wood [RW19] that, if a subprotocol for generating daBits is given, in which each daBit is a sharing of the same uniformly-random bit in both fields, except with negligible probability, then there exists a protocol that UC-securely realizes the functionality $\mathcal{F}_{\mathsf{Prep}}$. In this section, we give a new protocol for generating daBits to replace the procedure from [RW19]. The exposition focuses on the full-threshold case, since it is the most straightforward to explain. However, the protocol works for any Q2 access structure with some minor modifications, as described later.

The security of using the new protocol to realize our functionality $\mathcal{F}_{\mathsf{Prep}}$ essentially follows immediately from the proof in [RW19]; however, there are two minor differences between our $\mathcal{F}_{\mathsf{Prep}}$ and the corresponding functionality in [RW19], and so we also provide a sketch proof of the UC security. The differences are as follows. Firstly, we are only interested specifically in authenticated sharings of *field elements* in $\mathbb{F}_p$ whose bit-decomposition is also shared with authentication in $\mathbb{F}_2$ (not just bits). This is essentially a notational change since the conversion is entirely local, and so the functionalities are equivalent in the UC sense (i.e. there are UC reductions both ways). Secondly, Π_{daBits} refers explicitly to shares rather than abstract handles.

In Figure 9, we describe the protocol Π_{daShare} that produces double-authenticated shares $(\llbracket r \rrbracket_p, \llbracket \mathbf{r} \rrbracket_2)$, where $\llbracket r \rrbracket_p \leftarrow \sum_{k=0}^{\ell-1} 2^k \cdot \llbracket b^{(k)} \rrbracket_p$ and $\llbracket \mathbf{r} \rrbracket_2 = (\llbracket b^{(0)} \rrbracket_2, \ldots, \llbracket b^{(\ell-1)} \rrbracket_2)$. This protocol makes use of our new Π_{daBits} protocol (Figure 10) to produce ℓ daBits $(\llbracket b^{(k)} \rrbracket_p, \llbracket b^{(k)} \rrbracket_2)_{k=0}^{\ell-1}$. The proof of the following theorem is given later, after the proof of correctness of the new protocol Π_{daBits}.

Theorem 4.1. *The protocol Π_{daShare} UC-securely realizes $\mathcal{F}_{\mathsf{Prep}}$ in the presence of a static, active adversary in the $\mathcal{F}_{\mathsf{Prep}}[\mathsf{AuthLSSS}], \mathcal{F}_{\mathsf{Rand}}$-hybrid model.*

Π_{daShare}

daShare: Parties do the following:
1. Execute Π_{daBits} to generate ℓ daBits, $(\llbracket b^{(k)} \rrbracket_p, \llbracket b^{(k)} \rrbracket_2)_{k=0}^{\ell-1}$.
2. Compute $\llbracket r \rrbracket_p \leftarrow \sum_{k=0}^{\ell-1} 2^k \cdot \llbracket b^{(k)} \rrbracket_p$ and set $\llbracket \mathbf{r} \rrbracket_p \leftarrow (\llbracket b^{(k)} \rrbracket_2)_{k=0}^{\ell-1}$.
3. Output $(\llbracket r \rrbracket_p, \llbracket \mathbf{r} \rrbracket_2)$.

Figure 9. n-party daShare protocol

4.1 New daBit Protocol

The key observation leading to the new daBit protocol is that for a random bit shared in $\mathbb{F}_p$ between two parties, with probability $O(p^{-1})$ the XOR of the least significant bits (LSBs) of the two parties' shares is equal to the bit they share offset by $p \bmod 2$. Since typically $p = O(2^{\mathsf{sec}})$, this probability is generally negligible in the security parameter sec. This is proved in Lemma 4.1 . In the protocol, this observation is used to turn $n/2$ random bits in $\mathbb{F}_p$, secret-shared among n parties, into one daBit shared in both $\mathbb{F}_p$ and in $\mathbb{F}_2$.

Before giving the full protocol for generating daBits amongst n parties, we first provide the intuition. We emphasize that the focus here is on the full-threshold case, with necessary adaptations in the case of a Q2 access structure summarized later.
First, in the **Input** step, the parties generate $n/2$ random bits in $\mathbb{F}_p$. In the **DaBits Generation**, each of these is 'assigned' to one pair of adjacent parties as follows. For each

pair, $(2 \cdot i - 1, 2 \cdot i), i \in [n/2]$, the n-party sharing of one of the $n/2$ bits, $[\![b^{(i)}]\!]_p$, is turned into a 2-party sharing, which is achieved by each party other than $P_{2 \cdot i - 1}$ and $P_{2 \cdot i}$ sending their shares to one of these two parties. To decide which party sends to which in each pair, we define odd as the set of parties $\{P_1, P_3, \dots\}$ with odd index, and even as the set of parties $\{P_2, P_4, \dots\}$ with even index , and require parties always to open to the party with same parity as themselves. Thus for the adjacent pair $(P_{2 \cdot i-1}, P_{2 \cdot i})$, they obtain a sharing $\widehat{b}_i^{(2 \cdot i-1)} + \widehat{b}_i^{(2 \cdot i)} = b^{(i)} \bmod p$ of $b^{(i)}$. Now with high probability, it holds that $1 < \widehat{b}^{(2 \cdot i)}, \widehat{b}^{(2 \cdot i+1)} < p$, so $(\widehat{b}_i^{(2 \cdot i)} \bmod 2) \oplus (\widehat{b}_i^{(2 \cdot i+1)} \bmod 2) \oplus (p \bmod 2) = b_i \bmod 2$.

Now these two parties provide $\widehat{b}^{(2 \cdot i-1)} \bmod 2$ and $\widehat{b}^{(2 \cdot i)} \bmod 2$ as inputs for $\mathcal{F}_{\mathsf{Prep}}[\mathsf{AuthLSSS}]$ with *domain* $= \mathbb{F}_2$, so that the parties obtain an n-party boolean sharing of these values. Then all the parties subtract (XOR) the public value $p \bmod 2$ from this sharing in $\mathbb{F}_2$. Finally, after doing this for each pair, the parties take the XOR of all $n/2$ bits in $\mathbb{F}_2$ and in $\mathbb{F}_p$, which requires $n/2 - 1$ triples in $\mathbb{F}_p$. In the protocol, instead of removing the offset $p \bmod 2$ once for each pair, they remove it after the XOR with the offset $(n/2 \cdot (p \bmod 2)) \bmod 2$ to obtain the final daBit.

To achieve active security, i.e. to make sure parties are honest when providing inputs to $\mathcal{F}_{\mathsf{Prep}}[\mathsf{AuthLSSS}]$, the parties perform the **daBits Check** step. Note that to produce m correct daBits securely except with probability $2^{-\mathsf{sec}}$, the parties need to produce sec additional daBits that later will be discarded. This also means that the authentication of the 2-party bit is not necessarily correct when converting from an n-party sharing to a 2-party sharing since correctness is only guaranteed by the checking procedures.

As long as every party is involved in the generation of at least one daBit, any adversary corrupting up to $n-1$ parties will not learn the value of the daBit, as shown in the proof of Theorem 4.1. The protocol is given in Figure 10. It makes use of a subprotocol Π_{lsb}, of which we do not give a full description, that takes as input an authenticated secret-shared element of $\mathbb{F}_p$, $[\![x]\!]_p$ and returns the least significant bit of x in secret-shared form, $[\![b]\!]_p$. Protocols for this operation (with statistical security) can be found in the literature, e.g. [AKO+18, v1.5,§12.3.4]. It also uses a subprotocol Π_{xor}, which computes the (generalized) XOR function of a set of secret-shared bits in $\mathbb{F}_p$: given $\{[\![b^{(k)}]\!]_p\}_{k=1}^t$, it computes $[\![b]\!]_p \leftarrow [\![b^{(1)}]\!]_p$ and then iteratively computes $[\![b]\!]_p \leftarrow [\![b]\!]_p + [\![b^{(i)}]\!]_p - 2 \cdot [\![b]\!]_p \cdot [\![b^{(i)}]\!]_p$ for all $i = 2$ to t, requiring $t-1$ Beaver triples.

In the proof of Lemma 4.1, we argue that the daBits protocol is correct except with negligible probability in the statistical security parameter.

Lemma 4.1. *If the checks pass without parties aborting, then the protocol Π_{daBits} outputs correct daBits of uniformly random bits, except with probability $2^{-\mathsf{sec}}$, provided $m > 2 \cdot \mathsf{sec}$.*

Proof. First, we argue that the process of converting the n-party sharing to a 2-party sharing is correct. Given a sharing $\langle b \rangle_p$ of a secret bit, the two shares b_1 and b_2 are uniformly random in $\mathbb{F}_p$ subject to the constraint that $b_1 + b_2 \in \{0, 1\}$ and is 0 and 1 with equal probability. If $b_1 + b_2 = 0$ then the probability that they do *not* wrap around mod p is p^{-1} since the only way for this to happen is if $b_1 = b_2 = 0$. Thus $LSB(b_1) \oplus LSB(b_2) \oplus 1 = b_1 + b_2 \bmod p$ except with probability p^{-1}. If $b_1 + b_2 = 1$ then the probability that they do *not* wrap around mod p

Π_{daBits}

This protocol is realized in the $\mathcal{F}_{\mathsf{Prep}}[\mathsf{AuthLSSS}]$-hybrid model and uses the sub-protocols Π_{xor} and Π_{lsb}.

Input: A set of $\bar{m} = (m + \mathsf{sec}) \cdot n/2$ random authenticated bits in $\mathbb{F}_p$, $\{[\![b^{(k)}]\!]_p\}_{k\in[\bar{m}]}$, obtained by calling $\mathcal{F}_{\mathsf{Prep}}[\mathsf{AuthLSSS}].\mathsf{RandomBit}$.
daBits Generation: For each $k \in [m + \mathsf{sec}]$:

1. For each $i \in [n/2]$,
   (a) Each $P_j \in \mathsf{odd} \setminus \{P_{2\cdot i-1}\}$ sends their share $b_j^{(i+(k-1)\cdot n/2)}$ of $\langle b^{(i+(k-1)\cdot n/2)}\rangle_p$ to $P_{2\cdot i-1}$.
   Each $P_j \in \mathsf{even} \setminus \{P_{2\cdot i}\}$ sends their share $b_j^{(i+(k-1)\cdot n/2)}$ of $\langle b^{(i+(k-1)\cdot n/2)}\rangle_p$ to $P_{2\cdot i}$.
   (b) $P_{2\cdot i-1}$ sets $\widehat{b}_{2\cdot i-1}^{(k)} \leftarrow ((\sum_{j\in\mathsf{odd}} b_j^{(i+(k-1)\cdot n/2)} \bmod p) \bmod 2)$. $P_{2\cdot i}$ sets $\widehat{b}_{2\cdot i}^{(k)} \leftarrow ((\sum_{j\in\mathsf{even}} b_j^{(i+(k-1)\cdot n/2)} \bmod p) \bmod 2)$.
   (c) Parties call $\mathcal{F}_{\mathsf{Prep}}[\mathsf{AuthLSSS}]$ with *domain* $= \mathbb{F}_2$, and input $\widehat{b}_{2\cdot i-1}^{(k)}$ from $P_{2\cdot i-1}$, and input $\widehat{b}_{2\cdot i}^{(k)}$ from $P_{2\cdot i}$.
2. The parties set
$$[\![\widehat{b}^{(k)}]\!]_2 \leftarrow (\bigoplus_{i=1}^{n} [\![\widehat{b}_i^{(k)}]\!]_2) \oplus ((n/2 \cdot (p \bmod 2)) \bmod 2)$$
and
$$[\![\widetilde{b}^{(k)}]\!]_p \leftarrow \Pi_{\mathsf{xor}}(\{[\![b^{(i+(k-1)\cdot n/2)}]\!]_p\}_{i=1}^{n/2}).$$

daBits Check:

1. The parties call $\mathcal{F}_{\mathsf{Rand}}$ to obtain $m \cdot \mathsf{sec}$ random bits $\{\{\lambda_{k,\ell}\}_{k\in[m]}\}_{\ell\in[\mathsf{sec}]}$.
2. For each $\ell \in [\mathsf{sec}]$,the parties set
$$[\![\widetilde{c}]\!]_p \leftarrow \Pi_{\mathsf{lsb}}\left([\![\widetilde{b}^{(m+\ell)}]\!]_p + \sum_{k=1}^{m} \lambda_{k,\ell} \cdot [\![\widetilde{b}^{(k)}]\!]_p\right),$$
and
$$[\![\widehat{c}]\!]_2 \leftarrow [\![\widehat{b}^{(m+\ell)}]\!]_2 \oplus \bigoplus_{k=1}^{m} \lambda_{k,\ell} \cdot [\![\widehat{b}^{(k)}]\!]_2$$
3. Call $\mathcal{F}_{\mathsf{Prep}}[\mathsf{Auth}].\mathsf{Open}$ to obtain $\widetilde{c}$ and $\widehat{c}$, with *domain* $= \mathbb{F}_p$ and *domain* $= \mathbb{F}_2$, respectively.
4. If $\widetilde{c} \neq \widehat{c}$, then they abort.

Output If all sec checks have passed without the parties aborting then the parties output the first m sharings, $([\![\widetilde{b}^{(k)}]\!]_p, [\![\widehat{b}^{(k)}]\!]_2)_{k=1}^{m}$ as valid daBit sharings, and discard the rest.

Figure 10. n-party daBit generation

is the probability that exactly one of them is 1 and the other is 0, which happens in exactly two ways. Thus $LSB(b_1) \oplus LSB(b_2) \oplus 1 = b_1 + b_2 \bmod p$ except with probability $2 \cdot p^{-1}$. Overall, since the bit b is 0 or 1 with equal probability $\frac{1}{2}$, the probability that the protocol is is *not* correct is therefore $\frac{1}{2} \cdot p^{-1} + \frac{1}{2} \cdot 2 \cdot p^{-1} = \frac{3}{2} \cdot p^{-1} = O(2^{-\mathsf{sec}})$. Since the $n/2$-party XOR is computed from bits generated from the 2-party bits incurs an error of only a linear factor in n, and hence the probability is still negligible in sec.

Now suppose the checks all pass but there is an error in at least one daBit. Recall that a daBit $(\llbracket\widetilde{b}\rrbracket_p, \llbracket\widehat{b}\rrbracket_2)$ is considered correct if $\widetilde{b} = \widehat{b}$, and note that since the inputs are provided as input to the functionality $\mathcal{F}_{\mathsf{n-TinyOT}}$, the only possible error is to have $\widetilde{b} = \widehat{b} \oplus 1$ instead of $\widetilde{b} = \widehat{b}$ since only bits can be provided as input, and the secret $\widetilde{b}$ cannot be changed by the security of $\mathcal{F}_{\mathsf{Prep}}$ (and the subprotocols Π_{lsb} and Π_{xor}). Let $\boldsymbol{\varepsilon} \in \mathbb{F}_2^{m+\mathsf{sec}}$ be the error vector introduced by the adversary by changing inputs when calling $\mathcal{F}_{\mathsf{n-TinyOT}}$. If the checks pass, then for all $k \in [m+\mathsf{sec}]$ we can write $\widehat{b}^{(k)} = b^{(k)} \oplus \boldsymbol{\varepsilon}_k$ so that for every $\ell \in [\mathsf{sec}]$ it holds that

$$\begin{aligned}\left(\widetilde{b}^{(m+\ell)} + \sum_{k=1}^{m} \lambda_{k,\ell} \cdot \widetilde{b}^{(k)}\right) \bmod 2 &= \widehat{b}^{(m+\ell)} \oplus \bigoplus_{k=1}^{m} \lambda_{k,\ell} \cdot \widehat{b}^{(k)} \\ &= \widetilde{b}^{(m+\ell)} \oplus \boldsymbol{\varepsilon}_{m+\ell} \\ &\qquad \oplus \bigoplus_{k=1}^{m} \lambda_{k,\ell} \cdot (b^{(k)} \oplus \boldsymbol{\varepsilon}_k).\end{aligned}$$

Now the chance that $\boldsymbol{\varepsilon} \neq \mathbf{0}$ but that $\boldsymbol{\varepsilon}_{m+\ell} \oplus \bigoplus_{k=1}^{m} \lambda_{k,\ell} \cdot \boldsymbol{\varepsilon}_k = 0$ for a fixed ℓ is $\frac{1}{2}$, so because the sets $\{\boldsymbol{\lambda}_{k,\ell}\}_{k=1}^{m}$ are independently sampled for all $\ell \in [\mathsf{sec}]$, the probability that $\boldsymbol{\varepsilon} \neq \mathbf{0}$ but the above equation holds for all $\ell \in [\mathsf{sec}]$ is $2^{-\mathsf{sec}}$. The result follows. □

We are now ready to give a (sketch) proof of Theorem 4.1.

Proof (of Theorem 4.1 (Sketch)). We will assume that the subprotocol Π_{lsb} reveals no information about the secret. Since Π_{daShare} is a local procedure, it suffices to show that $\Pi_{\mathsf{daBits+MPC}}$ from [RW19] securely realizes $\mathcal{F}_{\mathsf{Prep}}$ when using our new protocol Π_{daBits} for generating daBits. To do this, we are required to show: firstly, that giving the adversary shares rather than handles does not give the environment a non-negligible distinguishing advantage; and secondly, that Π_{daBits} does not leak any information to the environment that may allow it to distinguish.

The argument for the first point is straightfoward: since the adversary corrupts at most all parties but one, every set of shares revealed to the adversary is indistinguishable from uniformly-random, and therefore can be simulated perfectly by the simulator given ideal outputs from $\mathcal{F}_{\mathsf{Prep}}$ by fixing the one honest party's input so that the sum is the correct output. Moreover, all shares handed to the adversary come from $\mathcal{F}_{\mathsf{Prep}}[\mathsf{AuthLSSS}]$ instantiated over $\mathbb{F}_p$ or $\mathbb{F}_2$, which the simulator emulates locally to $\mathcal{A}$, and so the transcripts in the hybrid world and ideal world are perfectly indistinguishable.

For the second point, observe that the proof in [RW19] showed that any leakage leading to a distinguishing advantage must come from the subprotocol generating daBits, Π_{daBits}, since the calls to $\mathcal{F}_{\mathsf{Prep}}[\mathsf{AuthLSSS}]$ (for each field) are black-box. Having argued that shares reveal nothing more than handles, we can argue in the same way.

Observe that for every daBit generated, there is at least one of the $n/2$ random bits generated in $\mathbb{F}_p$ for which there is at least one honest party whose share is never observed by the environment. Since the $n/2$-party XOR is computed, the sharing is uniform and unknown to the environment in this part of the subprotocol. Moreover, since the daBits $\{(\llbracket b_k \rrbracket_p, \llbracket \widehat{b_k} \rrbracket_2)\}_{k=m+1}^{m+\mathsf{sec}}$ are discarded, this removes all information about the linear relations amongst daBits learnt in the checking procedure as they can be viewed as keys for one-time-pad-encrypting the linear relations.

This means that even though each daBit generated by emulating honest parties in the hybrid world is different from the daBit generated by the functionality in the ideal world with probability $\frac{1}{2}$, the environment cannot observe this difference, and indeed by the high-level argument above, the simulator can always fix the share of the honest party so that outputs in the hybrid world simulation match the outputs in the ideal world.

The final argument that the daBit procedure does not leak information involves showing that the environment learns when the parties abort. A *selective failure attack* is one in which the environment can learn which world is being executed by attempting to cheat without detection. If it succeeds in doing so, it may be able to obtain enough information to distinguish between worlds. In our case, this means a corrupt party may cheat in providing its input into $\mathcal{F}_{\mathsf{Prep}}[\mathsf{AuthLSSS}]$: if the adversary guesses $\lambda_{k,\ell} = 0$ for some $k \in [m]$ ahead of time, then it can add an error to its share in $\mathbb{F}_p$, and if the checks do not abort then it will learn the final bit output from $\mathcal{F}_{\mathsf{Prep}}$ and with probability $\frac{1}{2}$ it will be 0, and thus the environment will know the interaction was ideal, not real. However, such an attack is mitigated by the fact that the check is repeated sec times, meaning that if cheating occurs then the parties abort with overwhelming probability in sec as argued in Lemma 4.1. Thus the environment cannot mount a selective failure attack successfully except with negligible advantage in sec, and hence we conclude the worlds are (statistically) indistinguishable to the environment. □

General Q2 Protocol In the case of an arbitrary Q2 access structure, when a bit is converted from an n-party sharing to a 2-party sharing, the pair must receive enough shares so that between them they hold an additive sharing of the secret, which typically requires less communication than in the full-threshold case. This is most easily done for a pair of parties (i, j) by taking a qualified set Q containing i and j and requiring all parties in Q to send their share(s) to one party or the other; then, using the recombination vector for Q, they can combine these shares into a two-party additive sharing.

For the secrecy of the final daBit (from the adversary, and the environment), it is not sufficient to generate $n/2$ bits and to turn them into 2-party sharings for pairs of *adjacent* parties as in the full-threshold case because it is possible that the adversary learns every bit during the conversion procedure. This happens (potentially) if in every pair, at least one party is corrupt and receives at least one share from an honest party.

Fortunately, there is an easy way to get around this problem by choosing the pairs so as to guarantee at least one pair exists for which both parties are honest. The following algorithm shows how to choose the pairs, denoted by a set S:

```
1. Set $S \leftarrow \varnothing$.
2. For every maximally-unqualified set $U$,
   (a) If $\nexists \{i,j\} \in S$ such that $\{i,j\} \not\subseteq [n] \setminus U$,
       i. Choose $\{i,j\} \subseteq [n] \setminus U$.
       ii. Set $S \leftarrow S \cup \{\{i,j\}\}$.
3. Output $S$.
```

The random choices can be made using a secure coin-flipping functionality. Thus for every maximally-unqualified set U, the set S contains a pair of parties not in U. Since the access structure is Q2, for whichever unqualified set of parties the adversary corrupts, there is always a pair of parties in which both parties are honest. While, in general, iterating over every maximally unqualified set potentially requires exponential computation in the number of parties, such iteration is already performed to establish the access structure in the first place so there is no additional (asymptotic) assumption on the required computational power.

5 Online Evaluation

Finally, we can give the online protocol Π_{MPC}(Figure 11) implementing the ideal functionality $\mathcal{F}_{\mathsf{MPC}}$ given in Figure 2 and Figure 3.

Protocol Π_{MPC}

Initialize: Parties call $\mathcal{F}_{\mathsf{Prep}}$ to activate the functionality.
Input, Evaluate, Output: There are two cases:
- If *domain* $= \mathbb{F}_p$, the parties run the sub-protocol $\Pi_{\mathsf{MPC}}[\mathsf{SPDZ}]$ on commands **Input, Evaluate, Output**
- Otherwise, the parties run the sub-protocol $\Pi_{\mathsf{MPC}}[\mathsf{BMR}]$ on commands **Input, Evaluate, Output**

Conversion: Parties call the sub-protocol $\Pi_{\mathsf{ConvertToField}}$ and $\Pi_{\mathsf{ConvertToBinary}}$

Figure 11. Zaphod online protocol

The sub-protocols $\Pi_{\mathsf{MPC}}[\mathsf{SPDZ}]$ (Figure 12) and $\Pi_{\mathsf{MPC}}[\mathsf{BMR}]$ (Figure 13) are the same as the SPDZ-online protocol [DPSZ12] and BMR-online protocol [HSS17], respectively. The only (minor) differences are:

- In $\Pi_{\mathsf{MPC}}[\mathsf{SPDZ}]$ we have the possibility to output to a single party, other than publicly to $\mathcal{P}$.
- As output of the BMR evaluation in $\Pi_{\mathsf{MPC}}[\mathsf{BMR}]$, the parties only obtain a sharing of the result of the computation. Note that this means that the final output of the online phase will always be a value in $\mathbb{F}_p$.

Sub-protocol $\mathit{\Pi}_{\mathsf{MPC}}[\mathsf{SPDZ}]$

Input: Given an input x enterred by party P_i:
1. The parties call the command (Input, P_i, r) of $\mathcal{F}_{\mathsf{Prep}}[\mathsf{SPDZ}]$, so that P_i obtains the *value* r, and the other parties obtain a handle on r, i.e. $[\![r]\!]_p$.
2. Party P_i broadcasts the value $y \leftarrow x - r$.
3. The parties compute $[\![x]\!]_p \leftarrow [\![r]\!]_p + y$.

Evaluate: - Add: On input $[\![x]\!]_p$, $[\![y]\!]_p$ the parties locally compute $[\![x+y]\!]_p \leftarrow [\![x]\!]_p + [\![y]\!]_p$.

- Multiply: On input $[\![x]\!]_p$, $[\![y]\!]_p$:
  1. The parties call *Triple* of $\mathcal{F}_{\mathsf{Prep}}[\mathsf{SPDZ}]$ to obtain a triple $([\![a]\!]_p, [\![b]\!]_p, [\![c]\!]_p)$.
  2. The parties compute the handles $[\![e]\!]_p \leftarrow [\![x]\!]_p - [\![a]\!]_p$ and $[\![f]\!]_p \leftarrow [\![y]\!]_p - [\![b]\!]_p$.
  3. The parties call $\mathcal{F}_{\mathsf{Prep}}[\mathsf{SPDZ}].\mathit{Open}$ on $[\![e]\!]_p$ and $[\![f]\!]_p$
  4. The parties evaluate $[\![z]\!]_p \leftarrow [\![c]\!]_p + e \cdot [\![b]\!]_p + f \cdot [\![a]\!]_p + e \cdot f$.

Output: To output a value $[\![y]\!]_p$ we distinguish two cases:
1. (**PublicOutput**):
   (a) Call $\mathcal{F}_{\mathsf{Prep}}[\mathsf{SPDZ}].\mathit{Check}$ on all opened handled/values since the last execution of $\mathcal{F}_{\mathsf{Prep}}[\mathsf{SPDZ}].\mathit{Check}$, if it aborts then abort.
   (b) Call $\mathcal{F}_{\mathsf{Prep}}[\mathsf{SPDZ}].\mathit{Open}$ on the handle $[\![y]\!]_p$.
   (c) Call $\mathcal{F}_{\mathsf{Prep}}[\mathsf{SPDZ}].\mathit{Check}$ on the single pair $([\![y]\!]_p, y)$ if it aborts then abort, otherwise accept y as the valid output.
2. (**PrivateOutput**):
   (a) The parties execute (Input, P_i, r) so that P_i obtains the *value* r, and the other parties obtain a handle on r, i.e. $[\![r]\!]_p$.
   (b) Call $\mathcal{F}_{\mathsf{Prep}}[\mathsf{SPDZ}].\mathit{Check}$ on all opened handled/values since the last execution of $\mathcal{F}_{\mathsf{Prep}}[\mathsf{SPDZ}].\mathit{Check}$, if it aborts then abort.
   (c) The parties compute $[\![x]\!]_p \leftarrow [\![r]\!]_p + [\![y]\!]_p$.
   (d) Call $\mathcal{F}_{\mathsf{Prep}}[\mathsf{SPDZ}].\mathit{Open}$ on the handle $[\![x]\!]_p$.
   (e) Call $\mathcal{F}_{\mathsf{Prep}}[\mathsf{SPDZ}].\mathit{Check}$ on the single pair $([\![x]\!]_p, x)$, if it aborts then abort. Otherwise, P_i computes and accepts $y \leftarrow x - r$.

Figure 12. The sub-protocol for realising evaluation over $\mathbb{F}_p$

Sub-protocol $\Pi_{\mathsf{MPC}}[\mathsf{BMR}]$

Input:
- The parties call $\mathcal{F}_{\mathsf{Prep}}[\mathsf{BMR}]$ on input $C_{\bar{f}}$ receiving all the wire masks $[\![\lambda_w]\!]_2$ and all the key $\mathbf{k}_{w,0}$.
- For each input wire w the parties have a shared value $[\![v_w]\!]_2$. The parties compute $[\![\epsilon_w]\!]_2 \leftarrow [\![v_w]\!]_2 \oplus [\![\lambda_w]\!]_2$. The value $[\![\epsilon_w]\!]_2$ is then opened to all parties.

Evalaute:
- For each input wire w P_i broadcasts the key $\mathbf{k}^i_{w,\epsilon_w}$.
- Each P_i now progresses through the circuit in the given topological ordering and locally processes the gates as follows:
  1. $c = \mathsf{XOR}(a, b)$ Gate:
     - $\epsilon_c \leftarrow \epsilon_a \oplus \epsilon_b$. and $\mathbf{k}^j_{c,\epsilon_c} \leftarrow \mathbf{k}^j_{a,\epsilon_a} \oplus \mathbf{k}^j_{b,\epsilon_b}$ for all $j \in [n]$.
  2. $c = \mathsf{INV}(a)$ Gate:
     - $\epsilon_c \leftarrow \epsilon_a \oplus 1$. and $\mathbf{k}^j_{c,\epsilon_c} \leftarrow \mathbf{k}^j_{a,\epsilon_a} \oplus \mathbf{k}^j_{\mathbf{one},1}$ for all $j \in [n]$.
  3. $c = \mathsf{AND}(a, b)$ Gate:
     - Recover $\mathbf{k}^j_{c,\epsilon_c}$ for all $j \in [n]$ by computing

$$\mathbf{k}^j_{c,\epsilon_c} \leftarrow g_{j,\epsilon_a,\epsilon_b} \oplus \Big(\bigoplus_{j=1}^{n} F_{\mathbf{k}^j_{a,\epsilon_a}, \mathbf{k}^j_{b,\epsilon_b}}(g \| j) \Big)$$

     - If $\mathbf{k}^i_{c,\epsilon_c} = \mathbf{k}^i_{c,0}$ set $\epsilon_c = 0$, if $\mathbf{k}^i_{c,\epsilon_c} = \mathbf{k}^i_{c,1}$ set $\epsilon_c = 1$, else *abort*.

Output: For every output wire w the parties compute $[\![v_w]\!]_2 \leftarrow [\![\lambda_w]\!]_2 \oplus \epsilon_w$, obtaining a (privately) share of $[\![v_w]\!]_2$.

Figure 13. The BMR-evaluation sub-protocol

5.1 The Conversion Protocol

All that remains to complete the online protocol is to give the protocols for *ConvertToField* and *ConvertToBinary* and show that their implementation does not break either the security of the LSSS based MPC components or the Garbled Circuit based MPC components. Given the functionality in Figure 8 our two conversion routines are trivial, see Figure 14 and Figure 15.

The most complex conversion is that in given in Figure 14 as it requires the evaluation of a garbled circuit. Here we utilize a simple circuit which takes as input the $\lceil \log_2 p \rceil$ bits of y, and the ℓ (secret shared) bits of r. The circuit computes $z = y + r \pmod{p}$, which can be done with a single addition and conditional subtraction. Then z is compared to $p/2$, if less than $p/2$ then the least 64 bits of z are extracted and assigned to x. Otherwise the circuit computes $w = p - z$, extracts the least 64 bits of w and then computes x the bitwise negation of these 64 bits plus one. As one can see this circuit involves only a small number of addition/subtraction operations, and is the main reason we restrict our selection of the prime p so that we do not have to perform more complex modular operations at this stage.

Theorem 5.1. *The protocol Π_{MPC} securely implements the functionality $\mathcal{F}_{\mathsf{MPC}}$ in the presence of a static active adversary corrupting up to $n-1$ parties in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model.*

Proof. Correctness follows by inspection of the protocol and by the choice of p. A simulator $\mathcal{S}$ for any adversary $\mathcal{A}$ works as follows. $\mathcal{S}$ emulates the adversary $\mathcal{A}$ corrupting the set $A \subset \mathcal{P}$ of parties and internally runs an execution of the protocol with dummy honest parties and adversary.

Sub-protocol $\Pi_{\mathsf{ConvertToBinary}}$

INPUT: A sharing $[\![x]\!]_p$

1. Set $\ell \leftarrow \lceil \log_2 p \rceil$ in the case that $64 + \mathrm{sec} \geq \lceil \log_2 p \rceil$, else set $\ell \leftarrow 64 + \mathrm{sec}$.
2. The parties call $\mathsf{DaShare}$ of $\mathcal{F}_{\mathsf{Prep}}$ with the given value of ℓ to get a doubly authenticated share $([\![r]\!]_p, [\![\mathbf{r}]\!]_2)$.
3. The parties compute $[\![y]\!]_p \leftarrow [\![x]\!]_p - [\![r]\!]_p$.
4. The value y is opened to all parties, so they all obtain $y = x - r$ in the clear.
5. The parties compute $[\![\mathbf{x}]\!]_2 \leftarrow y + [\![\mathbf{r}]\!]_2$ running the sub-protocol $\Pi_{\mathsf{MPC}}[\mathsf{BMR}]$ on a circuit representing addition modulo p. This circuit is designed to convert integers in the set $[0, \dots, 2^{63}) \cup (p - 2^{63}, p - 1]$ into 64-bit signed integers in two's complement representation.

Figure 14. Protocol to convert a $[\![\cdot]\!]_p$-sharing into a vector of 64 $[\![\cdot]\!]_2$-sharings

Sub-protocol $\Pi_{\mathsf{ConvertToField}}$

INPUT: A vector of sharings $[\![\mathbf{x}]\!]_2$

1. The parties call $\mathcal{F}_{\mathsf{Prep}}.\mathsf{DaShare}$ with $\ell = 64$, to obtain a doubly authenticated share $([\![r]\!]_p, [\![\mathbf{r}]\!]_2)$.
2. The parties locally compute, for $i = 0, \dots, 63$, the shares $[\![v_i]\!]_2 \leftarrow [\![x_i]\!]_2 \oplus [\![r_i]\!]_2$.
3. The value v_i is then opened to all parties.
4. The parties locally compute, for $i = 0, \dots, 63$, the shares $[\![x_i]\!]_p \leftarrow v_i + [\![r_i]\!]_p - 2 \cdot v_i \cdot [\![r_i]\!]_p$.
5. From these shared bits they can reconstruct x as an $[\![\cdot]\!]_p$-sharing via $[\![x]\!]_p \leftarrow -[\![x_{63}]\!]_p \cdot 2^{63} + \sum_{i=0}^{62} [\![x_i]\!]_p \cdot 2^i$.

Figure 15. Protocol to convert a vector of $[\![\cdot]\!]_2$-sharings into a $[\![\cdot]\!]_p$-sharing

1. INITIALIZE: The simulator emulates an activation of $\mathcal{F}_{\mathsf{Prep}}$
2. INPUT, EVALUATION, OUTPUT: The simulation of these steps is almost identical to the respective simulations of $\Pi_{\mathsf{MPC}}[\mathsf{SPDZ}]$ and $\Pi_{\mathsf{MPC}}[\mathsf{BMR}]$ in [DPSZ12] and [HSS17], respectively. In particular, first time *Input* is run, $\mathcal{S}$ broadcasts random values on behalf of honest parties and receives corrupt values from the adversary. The random masks used in this step are obtained from the emulation of $\mathcal{F}_{\mathsf{Prep}}$, hence they allow the extraction of corrupt parties' input. These inputs values are forwarded to the functionality $\mathcal{F}_{\mathsf{MPC}}$. Subsequent simulations of *Input* are done consistently with the emulation of the protocol, so honest shares and/or values are either computed by $\mathcal{S}$ honestly or sampled at random, depending on the different cases, and values from corrupt parties are received from $\mathcal{A}$. For self containment we include a sketch of the BMR-simulation:
   Input-BMR:
   - Receive from the adversary the following values 1) $\{\Delta_i\}_{i \in A}$, 2) for each input wire w associated to $P_i \in A$, $\{\lambda_{w,i}\}_{i \in A}$ and $\{k^i_{w,0}\}_{i \in A}$, 3) for each output wire w_{AND} of an AND gate, the keys $\{k^i_{w_{\mathsf{AND}},0}\}_{i \in A}$.
   - For each w_{AND}, sample a random mask λ_w
   - Compute $\lambda_c = \lambda_a + \lambda_b$ for each output wire of $\mathsf{XOR}(a, b) = c$
   - If receive *abort* from $\mathcal{A}$, forward *abort* to the functionality

   Evaluation-BMR:
   - For each input wire $\{w_i\}_{i \notin A}$ associated to a honest party P_i, sample a random public value $\{\epsilon_{w_i}\}_{i \notin A}$ and sent it to $\mathcal{A}$; for each input wire $\{w_i\}_{i \in A}$ associated to a corrupt party, receive the public values $\{\epsilon_{w_i}\}_{i \in A}$ from $\mathcal{A}$ and extract the inputs x_w.
   - Send those inputs to the functionality and receive the output y of the computation.

- Generate public values ϵ_w for each wire w in the circuit and compute the active path exactly as done in [HSS17].

3. Conversion: Both protocols $\Pi_{\mathsf{ConvertToField}}$ and $\Pi_{\mathsf{ConvertToBinary}}$ only consists of calling to the $\mathcal{F}_{\mathsf{Prep}}$ functionality and openings, so the simulation is straightforward.

Indistinguishability between real and ideal execution is proved similarly as in [HSS17] and [DPSZ12]. □

6 Engineering

Protocol	n	sec	Comm. (kb)	Throughput (ops/s)
daBit [RW19]	2	40	384	1008
dabit (ours)	2	40	94	2150
daBit [RW19]	3	40	1640	560
dabit (ours)	3	40	1104	650
daBit [RW19]	4	40	4781	306
dabit (ours)	4	40	2173	552
daBit [RW19]	2	80	931	424
daBit (ours)	2	80	288	706
daBit [RW19]	3	80	-	-
daBit (ours)	3	80	2442	285

Table 1. 10Gb/s LAN experiments for n-party daBit generation

In this section we outline a number of facets about how computing using both $[\![\cdot]\!]_p$ and $[\![\cdot]\!]_2$ sharings were integrated together into the `SCALE-MAMBA` system, which a system implementing LSSS-based MPC. To perform this integration a number of design decisions had to be made.

6.1 Threads and Data Queues

The `SCALE-MAMBA` system allows a number of independent MPC programs to run in parallel, thus creating a number of independent online 'threads'. Since pre-processing for the $\langle\cdot\rangle_p$-sharing based MPC is more expensive than the online processing, each online thread has a number of 'helper' threads which provide the pre-processed data. Thus for each thread implementing $\Pi_{\mathsf{MPC}}[\mathsf{SPDZ}]$ there are a four threads providing the functionalities of $\mathcal{F}_{\mathsf{Prep}}[\mathsf{SPDZ}]$ (for example Triples and Random Shared Bits). Each of the data types produced by the offline threads are added to queues, for consumption by the associated online thread. Keeping one queue per data type, per online thread, means there is no need for complex signaling between threads as would be needed if there was a single global queue for each data type.

In addition, there are a variable number of global threads (in the case of full threshold access structures) which perform the necessary ZKPoKs associated to the FHE-based ciphertexts (essentially these threads implement the HighGear/TopGear ZKPoKS [KPR18, BCS19]. In the default configuration there are two such ZKPoK threads. Thus if one is running o online threads then, in the default configuration for full threshold, one is actually running $4 \cdot o + 2$ actual threads.

This means that adding the garbled circuit functionality needs to be done in a way which does not degrade the performance of the existing $\langle \cdot \rangle_p$-based MPC routines too much. We thus settled for the following architecture. We added a single additional global thread (independent of the number of online threads), which performs the base OTs and OT-extensions, enabling the implementation of the procedure Get from $\mathcal{F}_{\mathsf{n-TinyOT}}$. This thread maintains a list of $o+1$ queues of authenticated bits $[\![b]\!]_2$, for random bits b. The throughput of such a single thread turns out to be more than enough to support the online operations in up to four online threads, since OT extension is so fast. The first o of the $o+1$ queues of authenticated bits are used to supply the online threads with authenticated bits, whereas the last of the authenticated bits queues is used to supply another thread whose task is to produce o queues of triples, one queue per online thread.

As a garbled circuit operation is encountered, the online thread polls the authenticated bit and triple thread for data; which is then supplied (if it is ready). When it is not yet ready (which happens if another GC operation has been performed recently, or if this is the first GC operation to be performed in a run) the online thread waits for the data to become available. When a garbled circuit needs to be evaluated, the garbling and evaluation are done on the fly by the online thread, resulting in authenticated bits and triples being consumed from the respective queues. We thus do not treat garbling as an “offline” operation.

The creation of daBits is performed within the online thread as and when needed. When a first request for daBits is encountered, the online thread produces a batch of daBits, consumes what it wants, and then stores any remaining for future use. Subsequent daBit requests either use the stored unused daBits, or if none are left more daBits are generated. When daBits are needed they are also consumed from the respective list.

Thus with this architecture we increase the number of actual threads from $4 \cdot o + 2$ to $4 \cdot o + 4$. By limiting the authenticated bit and triple producing threads to only produce a limited number of authenticated bits and triples, we can mitigate any slow-down of the online threads when no garbled circuit based operations are performed. However, we do get a small stall in performance when the first garbled circuit (resp. daBit) operation is met, and the associated queues of triples (resp. daBits) need to be filled.

6.2 Circuit Generation

By selecting to operate online on $\ell = 64$-bit data types, it means that we restrict the user to a small set of well defined operations on the 64-bit signed integers. Thus the user is not able to call arbitrary circuits, but has access to a library of circuits. This design decision is because most users are unable to design their own circuits, with few having either VHDL experience or access to expensive state-of-the-art tool chains. Thus to operate on 64-bit data types we extended the MAMBA language to support secure signed 64-bit arithmetic, with each arithmetic operation corresponding to a pre-defined combinatorial circuit. The combinatorial circuits were generated using the Synposis tool chain from a VHDL behavioural description of the desired circuit functionality.

However, a problem arises when using this methodology for the circuit needed in the $\mathbb{F}_2^\ell$ online evaluation. Users in SCALE-MAMBA can select their own prime p for the secret sharing scheme $\langle \cdot \rangle_p$, thus the circuit will depend not only on the bits of p, but also on the bit length. Thus it would seem that users need to be able to 'compile' their own circuit for such an operation, and thus would need access to tool chains such as that from Synopsis.

To avoid this issue we created a global circuit which performs the required operation for an arbitrary p (of bit length up to 512 bits). Then in a one-time setup procedure, this generic circuit, consisting of 6185 AND gates, 2893 XOR gates and 5865 INV gates, is specialized via a C-program to the specific prime selected by the user. This latter specialisation is then optimized further in C. The resulting circuit depends on the prime p being used, but for example in a 128-bit p variant the final optimized specialisation resulted in a circuit with 968 AND gates, 523 XOR gates and 629 INV gates.

We also extended SCALE-MAMBA to allow user-defined circuits to be executed. Thus, a user can define (say) a circuit for the Keccak round function and then evaluate it on 25 secure 64-bit integers (Keccak has a "block" size of 1600 bits). Combined with our daBit conversion routine this would allow the implementation of the distributed decryption algorithm in [KLO+19] completely within SCALE-MAMBA for an arbitrary number of parties. This algorithm requires switching between secret sharing modulo p, and then the resulting shared values need to be passed various garbled circuit constructions, and then passed back to secret sharing mod p.

6.3 Experiments

We benchmarked the daBit generation between computers with commodity hardware connected via a 10Gb/s LAN network with an average round-trip latency of 0.5ms. The results are presented in Table 1 with various statistical security parameters sec. According to the SCALE-MAMBA architecture, these numbers are produced using five FHE factories and eight online daBit generation threads.

The communication reported in [RW19] for statistical security sec = 40 in the two party case is 171kb which is about twice as low as the equivalent implementation in SCALE-MAMBA (384kb). The reason for this is that Rotaru and Wood estimated the pre-preprocessing per daBit using LowGear with every possible security parameter equal to 40 (zero knowledge proof soundness, distributed decryption, etc). The communication cost per triple used by them was 26.3kb. On the other hand we ran the daBit generation in parallel with TopGear which has a slightly higher communication cost per triple (78kb) but also enjoys higher security parameters for the ZK proof soundness (2^{-128}) and distributed decryption (2^{-40}). SCALE-MAMBA already had the method from [RW19] implemented within it, thus making an apples-to-apples comparison with our method was easier.

For the two party case we see that communication cost are reduced by a quarter, with the throughput doubled, using the new method for daBit generation. In the three party case the reduction in communication is less pronounced, and the throughput is roughly the same.

However for an odd number of parties n the new method has roughly the same complexity as the case for the $n+1$ parties. Thus when we move to four parties the improvement in the new method in terms of communication again returns. At 80-bit security we found it impossible to run the method from [RW19] for three parties, as the limits on the queues of pre-processed data in `SCALE-MAMBA` were too small for what is required by the method.

We have also benchmarked the total conversion cost of a 64 bit integer with statistical security $\mathrm{sec} = 40$ for the two party case. As expected, the conversion throughput from LSSS to GC is 104 times lower than generating a single daBit ending in 2.12MBytes per conversion (total communication) with a throughput of 20 conversions per second. The reason for a factor of 100 slowdown compared to the daBit generation is that a single conversion requires 104 daBits.

Acknowledgements

We would like to thank the anonyomous WAHC 2019 reviewers for their comments.

This work has been supported in part by ERC Advanced Grant ERC-2015-AdG-IMPaCT, 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 and FA8750-19-C-0502, by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA) via Contract No. 2019-1902070006, and by the FWO under an Odysseus project GOH9718N. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of any of the funders. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

References

AKO+18. Abdelrahaman Aly, Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Nigel P. Smart, and Tim Wood. SCALE-MAMBA v1.2: Documentation, 2018.

BCS19. Carsten Baum, Daniele Cozzo, and Nigel P. Smart. Using topgear in overdrive: A more efficient zkpok for SPDZ. *IACR Cryptology ePrint Archive*, 2019:35, 2019.

BDK+18. Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. HyCC: Compilation of hybrid protocols for practical secure computation. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 847–861, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

BDOZ11. Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188, Tallinn, Estonia, May 15–19, 2011. Springer, Heidelberg, Germany.

BGW88. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 1–10, Chicago, IL, USA, May 2–4, 1988. ACM Press.

BMR90. Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd Annual ACM Symposium on Theory of Computing*, pages 503–513, Baltimore, MD, USA, May 14–16, 1990. ACM Press.

BPSW07. Justin Brickell, Donald E. Porter, Vitaly Shmatikov, and Emmett Witchel. Privacy-preserving remote diagnostics. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS*

2007: 14th Conference on Computer and Communications Security, pages 498–507, Alexandria, Virginia, USA, October 28–31, 2007. ACM Press.

Can00. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. http://eprint.iacr.org/2000/067.

CCD88. David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 11–19, Chicago, IL, USA, May 2–4, 1988. ACM Press.

CGH$^+$18. Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. Cryptology ePrint Archive, Report 2018/570, 2018. https://eprint.iacr.org/2018/570.

CGR$^+$17. Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. EzPC: Programmable, efficient, and scalable secure two-party computation for machine learning. Cryptology ePrint Archive, Report 2017/1109, 2017. https://eprint.iacr.org/2017/1109.

DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

DSZ15. Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *ISOC Network and Distributed System Security Symposium – NDSS 2015*, San Diego, CA, USA, February 8–11, 2015. The Internet Society.

FKOS15. Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A unified approach to MPC with preprocessing using OT. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015, Part I*, volume 9452 of *Lecture Notes in Computer Science*, pages 711–735, Auckland, New Zealand, November 30 – December 3, 2015. Springer, Heidelberg, Germany.

GKWY19. Chun Guo, Jonathan Katz, Xiao Wang, and Yu Yu. Efficient and secure multiparty computation from fixed-key block ciphers. *IACR Cryptology ePrint Archive*, 2019:74, 2019.

HKS$^+$10. Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: tool for automating secure two-party computations. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM CCS 2010: 17th Conference on Computer and Communications Security*, pages 451–462, Chicago, Illinois, USA, October 4–8, 2010. ACM Press.

HSS17. Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 598–628, Hong Kong, China, December 3–7, 2017. Springer, Heidelberg, Germany.

KLO$^+$19. Michael Kraitsberg, Yehuda Lindell, Valery Osheter, Nigel P. Smart, and Younes Talibi Alaoui. Adding distributed decryption and key generation to a ring-LWE based CCA encryption scheme. In Julian Jang-Jaccard and Fuchun Guo, editors, *ACISP 19: 24th Australasian Conference on Information Security and Privacy*, volume 11547 of *Lecture Notes in Computer Science*, pages 192–210, Christchurch, New Zealand, July 3–5, 2019. Springer, Heidelberg, Germany.

KOS15. Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 724–741, Santa Barbara, CA, USA, August 16–20, 2015. Springer, Heidelberg, Germany.

KOS16. Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 830–842, Vienna, Austria, October 24–28, 2016. ACM Press.

KPR18. Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 158–189, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany.

KSS14. Florian Kerschbaum, Thomas Schneider, and Axel Schröpfer. Automatic protocol selection in secure two-party computations. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *ACNS 14: 12th International Conference on Applied Cryptography and Network Security*, volume 8479 of *Lecture Notes in Computer Science*, pages 566–584, Lausanne, Switzerland, June 10–13, 2014. Springer, Heidelberg, Germany.

KW93. Mauricio Karchmer and Avi Wigderson. On span programs. In *Proceedings of Structures in Complexity Theory*, pages 102–111, 1993.

MR18. Payman Mohassel and Peter Rindal. ABY3: A mixed protocol framework for machine learning. Cryptology ePrint Archive, Report 2018/403, 2018. https://eprint.iacr.org/2018/403.

NNOB12. Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 681–700, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

RW19. Dragos Rotaru and Tim Wood. Marbled circuits: Mixing arithmetic and boolean circuits with active security. *IACR Cryptology ePrint Archive*, 2019:207, 2019.

SW19. Nigel P. Smart and Tim Wood. Error detection in monotone span programs with application to communication-efficient multi-party computation. In Mitsuru Matsui, editor, *Topics in Cryptology – CT-RSA 2019*, volume 11405 of *Lecture Notes in Computer Science*, pages 210–229, San Francisco, CA, USA, March 4–8, 2019. Springer, Heidelberg, Germany.

WRK17. Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 39–56, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

Yao82. Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science*, pages 160–164, Chicago, Illinois, November 3–5, 1982. IEEE Computer Society Press.