MArBled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security

Dragos Rotaru[1,2] and Tim Wood[1,2]

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

Abstract. There are two main ways of performing computation on private data: one method uses linear secret-sharing, in which additions require no communication and multiplications require two secrets to be broadcast; the other method is known as circuit garbling, in which a circuit is somehow randomised by one set of parties and then evaluated by another. There are different advantages and disadvantages to each method in terms of communication and computation complexity. The main disadvantage of secret-sharing-based computation is that many non-linear operations require many rounds of communication. On the other hand, garbled circuit (GC) solutions require only constant rounds. Mixed protocols aim to leverage the advantages of both methods by switching between the two dynamically.
In this work we present the first mixed protocol secure in the presence of a dishonest majority for any number of parties and an active adversary. We call the resulting mixed arithmetic/Boolean circuit a marbled circuit[3]. Our implementation showed that mixing protocols in this way allows us to evaluate a linear Support Vector Machine with 400 times fewer AND gates than a solution using GC alone albeit with twice the preprocessing required using only SPDZ (Damgård et al., CRYPTO '12). When evaluating over a WAN network, our online phase is 10 times faster than the plain SPDZ protocol.

1 Introduction

One of the major modern uses of cryptography is for mutually-distrustful parties to compute a function on their combined secret inputs so that all parties learn the output and no party learns anything more about other parties' inputs than what can be deduced from their own input and the output alone. This is known as secure multiparty computation (MPC).

Recently, MPC has been shown to be very efficient for evaluating general Boolean [NNOB12,DZ13] and arithmetic [DPSZ12,DKL+13,KOS16,KPR18] circuits. Many real-world use cases of computing on private data involve some form of statistical analysis, requiring evaluation of arithmetic formulae. Perhaps the most common method of computing arithmetic circuits on private data is *secret-sharing*, in which secret inputs are split up into several pieces and distributed amongst a set of parties, which then perform computation on these shares, and recombine them at the end for the result.

Most modern MPC protocols for arithmetic circuits are designed in the preprocessing model, in which the computation is split into a computationally-expensive input-independent preprocessing phase called the *offline phase*, and an *online phase* which makes use of this preprocessed data but only uses information-theoretic primitives. Since the online phase is information-theoretically secure, the protocol as a whole is generally "as secure" as the offline phase. In the online phase of these protocols, additions usually come "for free" – meaning no communication amongst parties is required – and the multiplication of two secret-shared field elements requires only two messages to be broadcasted per party.

[3] See paper marbling.

MPC over a finite field or a ring is used to emulate arithmetic over the integers, and consequently, non-linear operations such as comparisons between secrets (i.e. $<, >, =$) are an important feature of MPC protocols. One of the shortcomings of MPC based on secret-sharing is that these natural but more complicated procedures require special preprocessing and several rounds of communication.

One way to mitigate these costs would be to use *circuit-garbling* instead of secret-sharing for circuits involving lots of non-linear operations, since this method has low (in fact, constant) round complexity. Recent work has shown that multiparty Boolean circuit garbling with active security in the dishonest majority setting can be made very efficient [WRK17, HSS17, KY18]. However, performing general arithmetic computations in Boolean circuits can be expensive since the arithmetic operations must be accompanied with reduction modulo a prime inside the circuit. Moreover, efficient constructions of multiparty constant-round protocols for *arithmetic* circuits remain elusive. Indeed, the best-known optimisations for arithmetic circuits such as using a primorial modulus [BMR16] are expensive even for passive security in the two-party setting. The only work of which the authors are aware in the multiparty setting is the passively-secure honest-majority work by Ben-Efraim [BE17].

So-called *mixed protocols* are those in which parties switch between secret-sharing (SS) and a garbled circuit (GC) mid-way through a computation, thus enjoying the efficiency of the basic addition and multiplication operations in any field using the former and the low-round complexity of GCs for more complex subroutines using the latter. One can think of mixed protocols as allowing parties to choose the most efficient field in which to evaluate different parts of a circuit.

Demmler et al. [DSZ15] constructed a mixed protocol with passive security in the two-party setting. In their work, known as ABY, they convert between arithmetic, Boolean, and Yao sharings. For small subcircuits, converting arithmetic shares to Boolean shares (of the bit decomposition) of the same secret – i.e. without any garbling – suffices for efficiency gains over performing the same circuits in with arithmetic shares; however, for large subcircuits, using garbling allows reducing online costs. Mohassel and Rindal [MR18] constructed a three-party protocol known as ABY3 for mixing these three types of sharing in the malicious setting assuming at most one corruption.

For mixed protocols to be efficient, clearly the cost of switching between secret-sharing and garbling, performing the operation, and switching back must be more efficient than the method that does not require switching, perhaps achieved by relegating some computation to the offline phase. For active security, it is additionally essential to retain authentication of secrets through the conversion. For secret-sharing-based MPC the *de facto* method of authentication is to use linearly-homomorphic information-theoretic MACs on all of the secrets. One of the key difficulties of creating mixed protocols in the active setting is that authenticating in this way requires that the conversion maintain authentication under MAC keys *in different fields* through the switch into and out of the circuit.

In this work we show how to do this – specifically, how to realise a **c**ircuit and **a**rithmetic **b**lack **b**ox (CABB), given by the functionality $\mathcal{F}_{\mathsf{CABB}}$ in Figure 1 – and thus we obtain an efficient mixed protocol in the full-threshold malicious setting. Our solution uses MPC in a black-box way and, while the circuit garbling requires small modification, is compatible with many state-of-the-art multiparty Boolean circuit garbling techniques, the only requirement being that parties should be able to authenticate their own choices of inputs, and that XOR can be computed between authenticated bits. (This is discussed in detail later.) As the garbling takes place in the $\mathcal{F}_{\mathsf{MPC}}$-hybrid model, we need make no restriction on the access structure, making it compatible with the recent generalized compilers for various access structures [ABF+18, AKO+18]. The usual functionality $\mathcal{F}_{\mathsf{MPC}}$ from [LPSY15, KY18] is given as part of the functionality $\mathcal{F}_{\mathsf{Prep}}$ in Figures 4 and 5.

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

The function $\mathsf{BitDec} : \mathbb{F} \to \{0,1\}^{\lfloor \log |\mathbb{F}| \rfloor}$ takes an element of $\mathbb{F}$ and computes its bit-decomposition as a sequence of bits. The functionality keeps track of the current session using a session identifier sid. If a party provides an input with sid different from what was sent in **Initialise**, the functionality outputs Reject to all parties and awaits another message.

Initialise On input $(\mathsf{Initialise}, \mathbb{F}, \mathsf{sid})$ from all parties, initialise a database of secrets Val with indices Val.Keys and store the field as $\mathsf{Val.Field} \leftarrow \mathbb{F}$.

Input On input $(\mathsf{Input}, i, \mathsf{id}, x, \mathsf{sid})$ from P_i and $(\mathsf{Input}, P_i, \mathsf{id}, \perp, \mathsf{sid})$ from all other parties, if $\mathsf{id} \notin \mathsf{Val.Keys}$ and $x \in \mathsf{Val.Field}$ then insert id into Val.Keys and set $\mathsf{Val}[\mathsf{id}] \leftarrow x$.

Add On input $(\mathsf{Add}, \mathsf{id}_x, \mathsf{id}_y, \mathsf{id}, \mathsf{sid})$, if $\mathsf{id}_x, \mathsf{id}_y \in \mathsf{Val.Keys}$ then set $\mathsf{Val}[\mathsf{id}] \leftarrow \mathsf{Val}[\mathsf{id}_x] + \mathsf{Val}[\mathsf{id}_y]$.

Multiply On input $(\mathsf{Multiply}, \mathsf{id}_x, \mathsf{id}_y, \mathsf{id}, \mathsf{sid})$, if $\mathsf{id}_x, \mathsf{id}_y \in \mathsf{Val.Keys}$ then set $\mathsf{Val}[\mathsf{id}] \leftarrow \mathsf{Val}[\mathsf{id}_x] \cdot \mathsf{Val}[\mathsf{id}_y]$.

Evaluate Circuit On input $(\mathsf{EvaluateCircuit}, C, \mathsf{id}_1, \dots, \mathsf{id}_t, \mathsf{id}, \mathsf{sid})$ where $t \cdot \log |\mathsf{Val.Field}|$ is the arity of Boolean circuit C, if $\mathsf{id}_i \in \mathsf{Val.Keys}$ for all $i \in [t]$, then await a message OK or Abort from the adversary. If it is Abort, then send Abort to all parties and halt, and otherwise set $\mathsf{Val}[\mathsf{id}] \leftarrow C(\mathsf{BitDec}(\mathsf{Val}[\mathsf{id}_1]), \dots, \mathsf{BitDec}(\mathsf{Val}[\mathsf{id}_t]))$ and continue.

Output On input $(\mathsf{Output}, \mathsf{id}, \mathsf{sid})$ from all parties, if $\mathsf{id} \in \mathsf{Val.Keys}$ then send $\mathsf{Val}[\mathsf{id}]$ to the adversary and await a response OK or Abort. If the message is OK, then send $\mathsf{Val}[\mathsf{id}]$ to all parties and continue, and otherwise send Abort and halt.

Fig. 1. Functionality $\mathcal{F}_{\mathsf{CABB}}$

1.1 Our Contribution

The main contribution of this work is a mixed protocol in the dishonest majority setting with active security. Our implementation shows that this is achievable with concrete efficiency.

When considering mixed protocols in the active setting, the primary technical challenge is in maintaining authentication through the transition from secret-shared inputs and secret inputs inside the GC, and *vice versa*. The naïve way of obtaining authentication from SS to GC is for parties to bit-decompose the shares of their secrets and the MACs locally and use these as input bits to the circuit, and validating inside the GC. This solution would require $O(n \cdot \kappa \cdot \log |\mathbb{F}|)$ bits per party to be sent to switch inputs in the online phase, where n is the number of parties, κ is the computational security parameter, and $\mathbb{F}$ is the MPC field, since each party needs to broadcast a GC key for each bit of the input. This method also requires garbling several additions and multiplications inside the circuit to check the MAC. The advantage of this solution, despite these challenges, is that it requires no additional preprocessing, nor adaptations to the garbling procedure.

Contrasting this approach, our solution makes use of special preprocessing to speed up the conversion. This results in reducing the circuit size by approximately $100,000$ AND gates per conversion for a field with a 128-bit prime modulus (assuming Montgomery multiplication is used). Let $\mathbb{F}_q$ denote the finite field of order q. In this work we show how to convert between secret-shared data in $\mathbb{F}_p$, where p is a large prime and is the MPC modulus, and GCs in $\mathbb{F}_{2^k}$ through the use of "**d**ouble-shared" **a**uthenticated **bits** which we dub *daBits*, following the nomenclature set out by [NNOB12]. These doubly-shared secrets are values in $\{0, 1\}$ shared and authenticated both in $\mathbb{F}_p$ and $\mathbb{F}_{2^k}$, where by 0 and 1 we mean the additive and multiplicative identity, respectively, in each field. In brief, the conversion of a SS input a into a GC involves constructing a random secret r in $\mathbb{F}_p$ using daBits, opening $a - r$ in MPC, bit decomposing this public value (requiring no communication) and using these as signal bits for the GC, and then in the circuit adding r and computing this modulo p, which is possible since the bit decomposition of r is doubly-shared. This keeps the authentication check mostly outside of the circuit instead requiring that the MAC on $a - r$ is correct. Going the other way around, the output of the circuit is a set of public signal bits whose masking bits are chosen to be daBits. To get the output, parties XOR the public signal bits with the $\mathbb{F}_p$ shares of the corresponding daBit masks, which can be done locally. These shares can then be used to reconstruct elements of $\mathbb{F}_p$ (or remain as bits if desired).

The only use of doubly-shared masks is at the two boundaries (input and output) between a garbled circuit and secret-sharing; all secrets used in evaluating arithmetic circuits (i.e. using standard SS-based MPC) are authenticated shares in $\mathbb{F}_p$ only; all wire masks "inside" the circuit (that is, for all wires that are *not* input or output wires) are authenticated shares of bits in $\mathbb{F}_{2^k}$ only. The *online* communication cost of our solution is that of each party broadcasting a single field element and then broadcasting $\log |\mathbb{F}|$ key shares per input, for a circuit of any depth. Thus the cost is $O(\kappa \log |\mathbb{F}|)$ per party, per field input to the circuit. The offline cost grows quadratically in n as generating daBits requires every party to communicate with every other party.

While the main focus of this work is to allow Boolean circuits to be evaluated on (the bits of) field elements of $\mathbb{F}_p$, our method gives a full arithmetic/Boolean/garbled circuit mixed protocol as once the bits of $a - r$ are public and the bit decomposition of r is known in $\mathbb{F}_{2^k}$, the parties can run the Boolean circuit computing $(a - r) + r \mod p$ to obtain the bits of a in the field $\mathbb{F}_{2^k}$ *with authentication*. Converting back to $\mathbb{F}_p$ involves XORing the public signal bits with shared daBits (which is free in $\mathbb{F}_{2^k}$). Our work is also compatible of converting classic SPDZ shares in $\mathbb{F}_p$ with the recent protocol SPDZ2k of Cramer et al [CDE$^+$18].

We remark that several of the multiparty arithmetic garbling techniques of [Ben17] require the use of "multifield shared bits", which precisely correspond to our daBits (albeit in an unauthenticated honest-majority setting), and consequently we suggest that our idea of generating daBits may lead to more efficient actively-secure multiparty garbling of arithmetic circuits.

Our construction involves two steps: the first extends the MPC functionality to allow for the same bits to be generated in two independent $\mathcal{F}_{\mathsf{MPC}}$ sessions in two different fields; the second uses this extended MPC functionality to perform the garbling SPDZ-BMR-style [LPSY15], which we explain in detail in Section 2. Thus, while replacing the garbling is not an entirely black-box procedure, the necessary modifications to existing protocols are modest.

Active security beyond bounded inputs While essentially all of the basic actively-secure MPC protocols enable the evaluation of additions and multiplications, for more complicated non-linear functions the only solutions that exist are those that require additional assumptions on the input data. For example, comparison requires bit decomposition, which itself requires that secrets be bounded by some constant. Since the bits of each input are directly inserted into the circuit, we can avoid this additional assumption. We refer the reader to [DFK$^+$06] or the documentation for the SCALE/MAMBA project [AKO$^+$18, §10 Advanced Protocols] for an overview of implementations of other functions in MPC.

2 Preliminaries

In this section we explain the basics of MPC and SDPZ-BMR. In our protocol, one instance of MPC is used to perform the secret-sharing-based MPC over a prime field, and another instance is used to perform circuit-garbling over a large field of characteristic 2.

2.1 General

We denote the number of parties by n, and the set of indices of parties corrupted by the adversary by A. We write $[j]$ to denote the set $\cup_{i=1}^{j}\{i\}$. We write $\mathbb{F}$ to denote a field, and $\mathbb{F}_q$ to denote the finite field of order q. The arithmetic circuit will be computed in the field $\mathbb{F}_p$ where p is a large prime, and the keys and masks for the garbled circuit in $\mathbb{F}_{2^k}$. By $\log(\cdot)$ we always mean the base-2 logarithm, $\log_2(\cdot)$. We denote by sec and κ the statistical and computational security parameters, respectively. We say that a function $\nu : \mathbb{N} \rightarrow \mathbb{R}$ is *negligible* if for every polynomial $f \in \mathbb{Z}[X]$ there exists $N \in \mathbb{N}$ such that $|\nu(X)| < |1/f(X)|$ for all $X > N$. If an event X happens with probability $1 - \nu(\mathsf{sec})$ where ν is a negligible function then we say that X happens with overwhelming probability in sec. We write $x \overset{\$}{\leftarrow} S$ to mean that x is sampled uniformly from the set S, and use $x \leftarrow y$ to mean that x is assigned the value y. We will sometimes denote by, for example, $(a-b)_j$ the j^{th} bit in the binary expansion of the integer $a - b$.

2.2 Security

UC Framework We prove our protocols secure in the universal composability (UC) framework of Canetti [Can00]. Protocols proved secure in this model are secure even when executed alongside arbitrarily many other protocols, concurrently, sequentially or both. We assume the reader's familiarity with this framework. In Figure 2 we give a functionality $\mathcal{F}_{\mathsf{Rand}}$ for obtaining unbiased random data that we need for our protocol. In Appendix A Figure 15 we provide the protocol Π_{Rand} that securely realises $\mathcal{F}_{\mathsf{Rand}}$ (without proof as it is standard and straightforward). As the instantiation of Π_{Rand} requires commitments, for completeness we also include the standard commitment functionality $\mathcal{F}_{\mathsf{Commit}}$ in Figure 16 and its protocol in Figure 17. We implicitly use a random oracle $\mathcal{F}_{\mathsf{RO}}$, given in Figure 18, since it is required for realising (our) UC commitments. The random oracle model is common in the MPC literature as its use gives efficient protocols and is not believed to lead to practical attacks.

Adversary We assume an active, static adversary corrupting at most $n-1$ out of n parties. An active adversary may deviate arbitrarily from the protocol description, and a static adversary is permitted to choose which parties to corrupt only at the beginning of the protocol, and cannot corrupt more parties later on. Our protocol allows corrupt parties to cause the protocol to abort before honest parties receive output, but if the adversary cheats then the honest parties will not accept an incorrect output. This is known as "security-with-abort" in the literature. Adversaries corrupting at most all parties but one are called "full-threshold". While this work focuses on the full-threshold setting, since $\mathcal{F}_{\mathsf{MPC}}$ is used as a black box, the access structure of our protocol is solely dependent on the access structure admitted by the instantiation(s) of $\mathcal{F}_{\mathsf{MPC}}$.

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

Random subset On input $(\mathsf{RSubset}, X, t)$ where X is a set satisfying $|X| \geq t$, sample $S \xleftarrow{\$} \{A \subseteq X : |A| = t\}$ and send S to all parties.

Random buckets On input $(\mathsf{RBucket}, X, t)$ where X is a set and $t \in \mathbb{N}$ such that $|X|/t \in \mathbb{N}$, set $n \leftarrow |X|/t$ and then for each $i = 1, \ldots, n$ do the following:

1. Sample $X_i \xleftarrow{\$} \{A \subseteq X : |A| = t\}$.
2. Set $X \leftarrow X \setminus X_i$.

Finally, send $(X_i)_{i=1}^{n}$ to all parties.

Fig. 2. Functionality $\mathcal{F}_{\mathsf{Rand}}$

Communication We assume point-to-point secure channels, and synchronous communication. Additionally, we assume a broadcast channel, which can be instantiated in the random oracle model over point-to-point secure channels[4]. A round of communication is a period of time in which parties perform computation and then send and receive messages. Messages sent in a round cannot depend on messages received during the current round, but messages may depend on messages sent in previous rounds. So-called *constant-round* protocols require $O(1)$ rounds for the entire protocol.

2.3 MPC

Our protocol makes use of MPC as a black box, with functionality outlined in Figure 4. The functionality $\mathcal{F}_{\mathsf{MPC}}$ over a field $\mathbb{F}$ is realised using protocols with statistical security sec if $|\mathbb{F}| = \Omega(2^{\mathsf{sec}})$ and computational security κ depending on the computational primitives being used. We will describe MPC as executed in the SPDZ-family of protocols [DPSZ12, DKL+13, KOS16, KPR18, CDE+18].

Secret-sharing A secret $x \in \mathbb{F}$ is said to be *additively shared* if a dealer samples a set $\{x^{(i)}\}_{i=1}^{n-1} \xleftarrow{\$} \mathbb{F}$, fixes $x^{(n)} \leftarrow x - \sum_{i=1}^{n-1} x^{(i)}$, and for all $i \in [n]$ sends $x^{(i)}$ to party P_i. Any set of $n-1$ shares is indistinguishable from a set of $n-1$ uniformly-sampled shares, and the sum of all n shares is the secret x. This secret-sharing is linear: the sum of corresponding shares of two secrets is a sharing of the sum of the two secrets, so no communication is required for linear functions.

There are three different types of shared value in our scheme, over two different fields, always additively shared. Additionally, information-theoretic MACs are used to ensure the adversary can cheat without detection with only negligible probability, which he does by correctly guessing the global MAC key. A secret $a \in \mathbb{F}_p$ is shared amongst the parties by additively sharing the secret a in $\mathbb{F}_p$ along with a linear MAC $\gamma_p(a)$ defined as $\gamma_p(a) \leftarrow \alpha \cdot a$, where $\alpha \in \mathbb{F}_p$ is a global MAC key, which is also additively shared. By "global" we mean that every MAC in the protocol uses this MAC key, rather than each party holding their own key and authenticating every share held by every other party. Similarly, a secret $c \in \mathbb{F}_{2^k}$ and its MAC $\gamma_{2^k}(c) = \Delta \cdot c$, where $\Delta \in \mathbb{F}_{2^k}$ is an additively-shared global MAC key, are additively shared in $\mathbb{F}_{2^k}$ amongst the parties.

[4] This can be done using a class of ε-almost universal hash functions as described in [DPSZ12, App. A.3]. In the setting of dishonest majority we cannot guarantee termination of a broadcast protocol, but honest parties will abort if there are significant delays between messages since synchronicity is assumed.

We denote shared, authenticated secrets in the following ways:

Sharing in $\mathbb{F}_p$	$[\![a]\!]_p = (a^{(i)}, \gamma_p(a)^{(i)}, \alpha^{(i)})_{i=1}^n$ where $a \in \mathbb{F}_p$ and $a^{(i)}, \gamma_p(a)^{(i)}, \alpha^{(i)} \in \mathbb{F}_p$ for all $i \in [n]$.
Sharing in $\mathbb{F}_{2^k}$	$[\![c]\!]_{2^k} = (c^{(i)}, \gamma_{2^k}(c)^{(i)}, \Delta^{(i)})_{i=1}^n$ where $c \in \mathbb{F}_{2^k}$ and $c^{(i)}, \gamma_{2^k}(c)^{(i)}, \Delta^{(i)} \in \mathbb{F}_{2^k}$ for all $i \in [n]$.
Sharing in both	$[\![b]\!]_{p,2^k} = ([\![b]\!]_p, [\![b]\!]_{2^k})$ where $b \in \{0, 1\}$.

The shares are considered correct if

$$\left(\sum_{i=1}^n \gamma_p(a)^{(i)}\right) = \left(\sum_{i=1}^n a^{(i)}\right) \cdot \left(\sum_{i=1}^n \alpha^{(i)}\right)$$

and

$$\left(\sum_{i=1}^n \gamma_{2^k}(c)^{(i)}\right) = \left(\sum_{i=1}^n c^{(i)}\right) \cdot \left(\sum_{i=1}^n \Delta^{(i)}\right)$$

and party P_i holds every value indexed by i. Moreover, secret $[\![b]\!]_{p,2^k}$ is considered correct if the bit is the same in both fields, by which we mean they are either both the additive identity or are both the multiplicative identity, in their fields. Creating these bits efficiently is one of the main contributions of this work. Notice that the superscript on the MACs is outside the bracket: the parties each hold one share of the MAC $\alpha \cdot a$ on a, not MACs on the shares $a^{(i)}$. The security of the MACs comes from the fact that, any adversary learns at most $n - 1$ values and so does not know the global MAC key and hence can only alter the secret and its MAC correctly with probability at most $1/|\mathbb{F}|$.

Addition of secrets Since the MAC is linear, authenticated addition is also local:

$$[\![a]\!] + [\![b]\!] \leftarrow [\![a + b]\!] = (a^{(i)} + b^{(i)}, \gamma_p(a)^{(i)} + \gamma_p(b)^{(i)}, \alpha^{(i)})_{i=1}^n$$

and similarly in $\mathbb{F}_{2^k}$. Multiplication by public constants follows immediately.

Addition of public values Addition of public values is also local:

$$[\![a]\!] + b \leftarrow \left((a^{(1)} + b, \gamma_p(a)^{(1)} + \alpha^{(1)} \cdot b, \alpha^{(1)}), (a^{(i)}, \gamma_p(a)^{(i)} + \alpha^{(i)} \cdot b, \alpha^{(i)})_{i=2}^n\right).$$

I.e. P_1 adds b to its share of a and every party adds $b \cdot \alpha^{(i)}$ to their share of the MAC. Observe that $\sum_{i=1}^n \alpha^{(i)} b = \alpha \cdot b$ so the MAC is correct.

Multiplication Using a technique due to Beaver [Bea92], multiplication of secrets can be computed as a *linear* operation of public values if the parties have access to some precomputed data. This data takes the form of so-called *Beaver triples* – triples of authenticated secrets $([\![a]\!], [\![b]\!], [\![a \cdot b]\!])$, where a and b are uniformly random and unknown to the parties. To multiply $[\![x]\!]$ and $[\![y]\!]$, the parties compute $[\![x - a]\!] \leftarrow [\![x]\!] - [\![a]\!]$ and $[\![y - b]\!] \leftarrow [\![y]\!] - [\![b]\!]$ locally and open them (i.e. they broadcast their shares of $[\![x - a]\!]$ and $[\![y - b]\!]$ so all parties learn $x - a$ and $y - b$), and then compute the product as

$$[\![x \cdot y]\!] \leftarrow [\![a \cdot b]\!] + (x - a) \cdot [\![b]\!] + (y - b) \cdot [\![a]\!] + (x - a) \cdot (y - b).$$

Since a and b are unknown to any party, $x - a$ and $y - b$ reveal nothing about x and y. We refer to protocols that generate Beaver triples as *SPDZ-like*; the generation of this input-independent data is called the *offline phase*, and the actual circuit evaluation the *online phase*. The main cost of SPDZ-style protocols comes from generating these Beaver triples. The two main ways of doing this are either using somewhat homomorphic encryption (SHE) [DPSZ12, DKL+13, KPR18] or oblivious transfer (OT) [KOS16, CDE+18].

Active security Since all operations in the online phase are linear, if we assume a secure preprocessing (offline) phase, in the online phase only *additive* errors need to be detected. This is where the MACs are used: if the MAC on the final output of the circuit being computed is incorrect, then an additive error has been introduced on the MAC or the secret, and in this case the parties abort. If there is no error, then either the output is correct, or (it can be shown that) the adversary must have learnt enough information to guess the global MAC key. If p is $O(2^{\mathsf{sec}})$ then the chance of the adversary doing so is already negligible, and otherwise parties can generate $\lceil \mathsf{sec}/\log p \rceil$ independent global MAC keys, hold this number of MACs on each secret and require that all MACs on the final output be correct. We refer the reader to [DKL+13] for details on the MAC checking procedure.

Conditions on the secret-sharing field Let $l = \lfloor \log p \rfloor$. Throughout, we assume the MPC is over $\mathbb{F}_p$ where p is some large prime, but we require that one must be able to generate uniformly random field elements by sampling bits uniformly at random $\{[\![r_j]\!]_p\}_{j=0}^{l-1}$ and summing them to get $[\![r]\!]_p \leftarrow \sum_{j=0}^{l-1} 2^j \cdot [\![r_j]\!]_p$. For this to hold in $\mathbb{F}_p$, we require that $\frac{p-2^l}{p} = O(2^{-\mathsf{sec}})$. Roughly speaking this says that p is slightly larger than a power of 2. (By symmetry of this argument we can require that p be close to a power of 2.) Recall that sampling a uniform element of $\{0,1\}^l$ produces the same distribution as sampling l bits independently by standard measure theory. It follows from Lemma 1 that the statistical distance between the uniform distribution over $\mathbb{F}_p$ and the same over $\{0,1\}^l$ is negligible.

Lemma 1. *Let $l = \lfloor \log p \rfloor$, let P be the probability mass function for the uniform distribution $\mathcal{P}$ over $[0, p) \cap \mathbb{Z}$ and let Q be the probability mass function for the uniform distribution $\mathcal{Q}$ over $[0, 2^l) \cap \mathbb{Z}$. Then the statistical distance between distributions is negligible in the security parameter if $\frac{p-2^l}{p} = O(2^{-\mathsf{sec}})$.*

Proof. By definition of statistical distance,

$$\begin{aligned}\Delta(\mathcal{P}, \mathcal{Q}) = \frac{1}{2} \cdot \sum_{x=0}^{p-1} |P(x) - Q(x)| &= \frac{1}{2} \cdot \sum_{x=0}^{2^l-1} \left| \frac{1}{p} - \frac{1}{2^l} \right| + \frac{1}{2} \cdot \sum_{x=2^l}^{p-1} \left| \frac{1}{p} - 0 \right| \\ &= \frac{1}{2} \cdot 2^l \cdot \frac{p - 2^l}{p \cdot 2^l} + \frac{1}{2} \cdot p - 2^l) \cdot \frac{1}{p} = \frac{p - 2^l}{p} = O(2^{-\mathsf{sec}}).\end{aligned}$$

Arbitrary Rings vs Fields Our protocol uses actively-secure MPC as black box, so there is no reason the MPC cannot take place over any ring $\mathbb{Z}/m\mathbb{Z}$ where m is possibly composite, as long as m is (close to) a power of 2. The security of our procedure for generating daBits can tolerate zero-divisors in the ring, so computation may, for example, take place over the ring $\mathbb{Z}/2^l\mathbb{Z}$ for any l, for which actively-secure $\mathcal{F}_{\mathsf{MPC}}$ can be realised using [CDE+18].

Notation The functionalities sometimes refer to identifiers for variables such as id, where the value inside is $\mathsf{Val}[\mathsf{id}]$. We will use $[\![x]\!]$ to denote the variable identifier id for the value x, so $\mathsf{Val}[[\![x]\!]] = x$, and saying that the parties have $[\![x]\!]$ does not imply they know the secret x. This intentional collision of notation for authenticated secrets is to demonstrate that the realisation of the dictionary Val occurs via the secret sharing with MACs. To save on overloaded notation, we will occasionally write $[\![z]\!] \leftarrow [\![x]\!] + [\![y]\!]$ to mean that parties send the command $(\mathsf{Add}, [\![x]\!], [\![y]\!], [\![z]\!])$ to some functionality, and similarly for multiplication. Where not explicitly specified, new identifiers are taken from a counter which the parties initialise to 0 at the start of the protocol.

Note on XOR In our context, we will require heavy use of the (generalised) XOR operation. This can be defined in any field as the function

$$\begin{aligned} f : \mathbb{F}_p \times \mathbb{F}_p &\to \mathbb{F}_p \\ (x, y) &\mapsto x + y - 2 \cdot x \cdot y, \end{aligned} \tag{1}$$

which coincides with the usual XOR function for fields of characteristic 2. In SS-based MPC, addition requires no communication, so computing XOR in $\mathbb{F}_{2^k}$ is for free; the cost in $\mathbb{F}_p$ (char$(p) > 2$) is one multiplication, which requires preprocessed data and some communication. This operation is the main cost associated with our offline phase, since generating daBits with active security requires generating lots of them and then computing several XORs in both fields.

2.4 Garbled Circuits

In this section we describe circuit garbling in the multiparty setting.

SPDZ-BMR Garbling Lindell et al. [LPSY15] gave generic multiparty method, known as SPDZ-BMR, for garbling in a constant number of rounds with malicious security where the preprocessed material is obtained from SPDZ [DPSZ12]. Their method is (roughly) to execute the classic [BMR90] multiparty garbling protocol using SPDZ to generate all the necessary secrets and to compute the ciphertexts. While the circuits are Boolean, the wires masks are arithmetic shares in $\mathbb{F}_p$ of binary values and the wire keys random elements of $\mathbb{F}_p$ secret-shared amongst the parties. Importantly, it was shown that it was not necessary for parties to provide zero-knowledge proofs that the evaluations of the pseudorandom function (PRF) used for encryption was done honestly, as the evaluators would abort with overwhelming probability in κ if parties cheated in this way.

The FreeXOR garbling technique [KS08] is an optimisation of Yao's original garbling [Yao86, Oral presentation] that requires no data to be sent between the garbler and evaluator for an XOR gate, and crucially relies on the fact that the keys are elements of a field of characteristic 2. Towards the goal of a multiparty garbling protocol with FreeXOR, one might hope to perform SPDZ-BMR over $\mathbb{F}_{2^k}$. One of the reasons this was not considered for SDPZ-BMR was (presumably) that the SPDZ offline phase was much faster for large prime fields than extension fields. Indeed, the most efficient variant of SPDZ used BGV [BGV11] as the SHE scheme, which meant that while the offline phase could parallelise through ciphertext packing, for large extension fields – and in particular for finite extensions of $\mathbb{F}_2$ – the amount of available packing was limited. However, shortly after this Keller et al. [KOS16] showed how to use oblivious transfer (OT) to perform the offline phase even more efficiently. This solution was shown to be more efficent than using SHE for extension fields. Despite recent work [KPR18] showing that SHE solutions outperform OT solutions for large prime fields, [KOS16] remains faster over extension fields. Subsequently, Keller and Yanai [KY18] showed how to apply FreeXOR in the multiparty setting using SPDZ-BMR-style garbling where the SPDZ shares are in $\mathbb{F}_{2^k}$ instead of $\mathbb{F}_p$.

Meanwhile, Hazay et al. [HSS17] also showed how to obtain FreeXOR in the multiparty setting, again over $\mathbb{F}_{2^k}$, but take a different approach from SPDZ-BMR: they do not make use of a a full-blown MPC functionality and instead produce an *unauthenticated* garbled circuit – it is merely additively shared, whereas in SPDZ-BMR and [KY18], the garbled circuit is authenticated with MACs. Active security comes from the fact that an incorrectly-garbled circuit will only cause the parties to abort when evaluating it. This approach requires only a single (authenticated) $\mathbb{F}_2$ multiplication per AND gate.

We use the multiparty Boolean circuit garbling protocol [KY18] for our implementation. The [KY18] protocol is less efficient than [HSS17] and [WRK17], but the implementation [Ana19] is easier to integrate with the SPDZ compiler to be able to switch between different online phases of an MPC program [ABF$^+$18]. Despite this we have good reason to believe that the generation of the specialised preprocessing required in our solution dovetails with most if not all of these alternative these garbling schemes as the only requirements are the following:

- Parties should be able to authenticate their own secret inputs (in fact, secret bits suffices), for whatever authentication method is used in the protocol.
- Parties should be able to compute the XOR of authenticated bits.

Unfortunately, the authentication is usually abstracted away garbling functionalities so we cannot make straightforward claims about using garbling in a black-box way.

Encryption Before we describe the garbling, we first briefly describe the encryption scheme used in the protocol. The formalism of the security of using FreeXOR in the multiparty context was not given in [KY18] so we provide an overview here based on [HSS17]. In the garbling protocol, messages are encrypted by computing the XOR of the message with a pseudorandom one-time-pad generated by a PRF under keys held by multiple parties. The SPDZ-BMR technique of encryption requires a PRF which is a *pseudorandom function under multiple keys* [LPSY15, Defn 1] (see Definition 2 in Appendix B). In order to use the FreeXOR technique, a stronger assumption is needed: *circular* 2*-correlation robust*. Detailed analysis of a similar assumption for hash functions was given by [CKKZ12], and Hazai et al. [HSS17] gave a variant for PRFs, of which we now give an outline.

To make the definition, we define the following oracles. Sample $R \xleftarrow{\$} \{0,1\}^\kappa$ and let $|C|$ denote the number of gates in the circuit. For a PRF $F : \{0,1\}^\kappa \times \{0,1\}^\kappa \times \{0,1\}^{\log |C| + \log n} \to \{0,1\}^\kappa$, define two oracles which take inputs from the set $\{0,1\}^\kappa \times \{0,1\}^\kappa \times [|C|] \times [n] \times \{0,1\}^3$:

- Oracle Circ_R: on input $(\mathsf{k}_1, \mathsf{k}_2, g, j, b_1, b_2, b_3)$, return $F_{\mathsf{k}_1 \oplus b_1 \cdot R, \mathsf{k}_2 \oplus b_2 \cdot R}(g \| j)) \oplus b_3 \cdot R$.
- Oracle RO: on input $(\mathsf{k}_1, \mathsf{k}_2, g, j, b_1, b_2, b_3)$, if the message has been queried before, output whatever was given last time; otherwise, sample a random string $\mathsf{k}_3 \xleftarrow{\$} \{0,1\}^\kappa$ and output k_3.

Definition 1 (See [HSS17, Defn 2.3]). *A PRF F is called* circular 2-correlation robust *if for any non-uniform polynomial-time distinguisher $\mathcal{D}$ making legal queries (see below) to its oracle, there exists a negligible function ν such that*

$$\left| \Pr[R \xleftarrow{\$} \{0,1\}^\kappa; \mathcal{D}^{\mathsf{Circ}_R(\cdot)}(1^\kappa) = 1] - \Pr[\mathcal{D}^{\mathsf{RO}(\cdot)}(1^\kappa) = 1] \right| \leq \nu(\kappa).$$

"Legal queries" are defined as any query except the following:

- $(\mathsf{k}_1, \mathsf{k}_2, g, j, 0, 0, b_3)$ (otherwise the distinguisher can distinguish trivially as it learns $F_{\mathsf{k}_1,\mathsf{k}_2}(g\|j)$ and it can compute this on its own).
- The query $(\mathsf{k}_1, \mathsf{k}_2, g, j, b_1, b_2, 1 - b_3)$ if the query $(\mathsf{k}_1, \mathsf{k}_2, g, j, b_1, b_2, b_3)$ has already been made (otherwise the global difference is leaked).

The encryption of a message $m \in \{0,1\}^\kappa$ under keys $\mathsf{k}_u \leftarrow \mathsf{k}_u^1\| \cdots \|\mathsf{k}_u^n$ and $\mathsf{k}_v \leftarrow \mathsf{k}_v^1\| \cdots \|\mathsf{k}_v^n$ and nonce r, where P_i holds they keys $\mathsf{k}_u^i, \mathsf{k}_v^i \in \{0,1\}^\kappa$, is defined as

$$\mathsf{Enc}_{\mathsf{k}_u,\mathsf{k}_v}(m; r) \leftarrow \left(\bigoplus_{j=1}^{n} F_{\mathsf{k}_u^j,\mathsf{k}_v^j}(r)\right) \oplus m.$$

For clarity, we will write the formula explicitly in the circuit garbling rather than abstract to the encryption notation.

Garbling Protocol The high-level view is as follows. Let $g : \{0,1\}^2 \to \{0,1\}$ denote the gate g. In circuit garbling, first the garbler samples a a "zero key" and a "one key" for every wire in the circuit. A wire connection exiting one gate and entering another is considered one wire, as are all circuit input and output wires. Then, each Boolean fan-in-two gate with input wires u, v and output wire w is converted to a set of 4 ciphertexts in the following way: for each $(\alpha, \beta) \in \{0,1\}^2$ the garbler encrypts the key $\mathsf{k}_{w,g(\alpha,\beta)}$ under the pair of input keys $\mathsf{k}_{u,\alpha}$ and $\mathsf{k}_{v,\beta}$. The garbler converts all gates to quadruples of ciphertexts. Note that this can be done in parallel for all gates. To evaluate the circuit, the evaluator is given the circuit input wire keys corresponding to his inputs (obliviously), and using these can decrypt one ciphertext in each quadruple (gate) to obtain the final output. Other measures, as we describe below, are required to hide information about intermediate wire values as the circuit is being garbled, but this is the basic outline.

In BMR garbling, every party acts as both garbler and evaluator. Each party generates a circuit for which it knows all the wire keys, but where each ciphertext is encrypted under *all* parties' corresponding wire keys. This means that every party must evaluate all n circuits in parallel to decrypt each subsequent gate and so learn all n succeeding keys. The idea of SPDZ-BMR garbling is to use MPC to compute the ciphertexts. The full protocol, modified for our purposes and incorporating FreeXOR, is provided in Figures 10, 11, 12 and 13, but we give an overview of correctness here. The functionality $\mathcal{F}_{\mathsf{MPC}}$ is part of $\mathcal{F}_{\mathsf{Prep}}$ in Figure 4. The parties first call $\mathcal{F}_{\mathsf{MPC}}$ with input $(\mathsf{Initialise}, \mathbb{F}_{2^k}, 1)$ where 1 is a session identifier.

Global difference Once at the beginning of the protocol, for each $i \in [n]$ the parties call $\mathcal{F}_{\mathsf{MPC}}$ with input $(\mathsf{RElt}, [\![R^i]\!]_{2^k}, 1)$ and then $(\mathsf{Open}, i, [\![R^i]\!]_{2^k}, 1)$ so that P_i obtains a random $R^i \in \mathbb{F}_{2^k}$, called its "global difference". These are used later for defining keys in a special way.

Wire Masks For each wire in the circuit, the parties call $\mathcal{F}_{\mathsf{MPC}}$ with input $(\mathsf{RBit}, [\![\lambda_w]\!]_{2^k}, 1)$. These are known as the *masking* or *permutation* bits and are used to permute the four ciphertexts in each gate, which is necessary to hide intermediate wire values (which leak information on the circuit inputs). Instead of evaluating $g(\alpha, \beta)$ on the actual inputs ρ and σ, evaluators hold "signal bits" $\Lambda_u \leftarrow \rho \oplus \lambda_u$ and $\Lambda_v \leftarrow \sigma \oplus \lambda_v$ and compute $\Lambda_w \leftarrow \tilde{g}(\Lambda_u, \Lambda_v)$ where $\tilde{g}$ is defined as $(\alpha, \beta) \mapsto g(\alpha \oplus \lambda_u, \beta \oplus \lambda_v) \oplus \lambda_w$ where λ_w is the mask for the output wire[5].

[5] This is equivalent to randomly permuting the four ciphertexts (indexed by $\{1,2,3,4\}$) by the (secret) permutation $((13)(24))^{\lambda_u}((12)(34))^{\lambda_v}$.

Garbling

- For an AND gate g with input wires u and v and output wire w, for each $i \in [n]$ the parties call $\mathcal{F}_{\mathsf{MPC}}$ with input $(\mathsf{RElt}, [\![\mathsf{k}^i_{u,0}]\!]_{2^k}, 1)$ to obtain a random field element and open it to party P_i. The one key $\mathsf{k}^i_{u,1}$ is set to be $\mathsf{k}^i_{u,0} \oplus R^i$ by P_i and the parties compute $[\![\mathsf{k}^i_{u,1}]\!]_{2^k} \leftarrow [\![\mathsf{k}^i_{u,0}]\!]_{2^k} \oplus [\![R^i]\!]_{2^k}$. The parties do the same for v and w. It is possible to garble with random one keys as described in the overview above, but this global difference allows more efficient garbling and evaluation as outlined below with no loss to security. Then each party evaluates the PRF on the four combinations of their own input keys $\{(\mathsf{k}^i_{u,\alpha}, \mathsf{k}^i_{v,\beta}) : \alpha, \beta \in \{0,1\}\}$ and calls $\mathcal{F}_{\mathsf{MPC}}$ with input

$$\left(\mathsf{Input}, i, [\![F^{g,i,j}_{\alpha,\beta}]\!]_{2^k}, F_{\mathsf{k}^i_{u,\alpha},\mathsf{k}^i_{v,\beta}}(g\|j), 1\right)$$

which form $4n$ authenticated (pseudorandom one-time-pad) encryption keys, indexed by $\alpha, \beta \in \{0,1\}$ and $j \in [n]$. In MPC, the parties then compute, for all $(\alpha,\beta) \in \{0,1\}^2$ and all $j \in [n]$, the ciphertext

$$[\![\tilde{g}^j_{\alpha,\beta}]\!]_{2^k} \leftarrow \left(\bigoplus_{i=1}^{n} \left[\!\!\left[F^{g,i,j}_{\alpha,\beta}\right]\!\!\right]_{2^k}\right) \oplus [\![\mathsf{k}^j_{w,0}]\!]_{2^k} \oplus [\![R^j]\!]_{2^k} \cdot (([\![\lambda_u]\!]_{2^k} \oplus \alpha) \cdot ([\![\lambda_v]\!]_{2^k} \oplus \beta) \oplus [\![\lambda_w]\!]_{2^k}) \,.$$

In other words, for each $j \in [n]$ the parties compute an encryption in MPC of wire key $k^j_{w,0} \oplus R^j \cdot ((\lambda_u \oplus \alpha) \cdot (\lambda_v \oplus \beta) \oplus \lambda_w)$ under all four possible pairs of input keys of every other party. Note that the j^{th} set of four ciphertexts are permuted by the *same* masks for every j.
- For an XOR gate, the parties set $\mathsf{k}^j_{w,0} \leftarrow \mathsf{k}^j_{u,0} \oplus \mathsf{k}^j_{v,0}$ for all j and the output mask as $\lambda_w \leftarrow \lambda_u \oplus \lambda_v$.

After all the garbling is performed, all the ciphertexts (currently held in the MPC engine) are opened.

Input For a party P_i to provide an input $x \in \{0,1\}$ on wire w, the parties open $[\![\lambda_w]\!]_{2^k}$ to P_i and then P_i broadcasts $\Lambda_w \leftarrow x \oplus \lambda_w$, known as a *signal bit*. For all $j \in [n]$, P_j broadcasts k^j_{w,Λ_w}.

Evaluation After the n keys and signal bits, one for each input wire, are obtained, the parties do the following.

- For an AND gate, for every $j \in [n]$, each party computes the n succeeding wire keys as

$$\begin{aligned} \mathsf{k}^j_{w,\cdot} &\leftarrow \tilde{g}^j_{\Lambda_u,\Lambda_v} \oplus \bigoplus_{i=1}^{n} F_{\mathsf{k}^i_{u,\Lambda_u},\mathsf{k}^i_{v,\Lambda_v}}(g\|j) \\ &= \mathsf{k}^j_{w,0} \oplus R^j \cdot ((\lambda_u \oplus \Lambda_u) \cdot (\lambda_v \oplus \Lambda_v) \oplus \lambda_w) \end{aligned}$$

Then each P_i compares $k^i_{w,\cdot}$ to the two keys $\mathsf{k}^i_{w,0}$ and $\mathsf{k}^i_{w,1}$ that it owns in order to determine the signal bit Λ_w. By the security of the PRF (as argued in SPDZ-BMR), the ciphertexts corresponding to the other keys cannot be decrypted. Observe that since $\Lambda_u = x \oplus \lambda_u$ and $\Lambda_v = y \oplus \lambda_v$ where x and y are the actual inputs, the resulting keys (for $j \in [n]$ are $\mathsf{k}^j_{w,0} \oplus R^j \cdot (x \cdot y \oplus \lambda_w)$, which are exactly $\mathsf{k}^j_{w,\Lambda_w}$ where $\Lambda_w = x \cdot y \oplus \lambda_w$.
- For an XOR gate, every party computes the n output signal bit as $\Lambda_w \leftarrow \Lambda_u \oplus \Lambda_v$ and sets the keys as $\mathsf{k}^j_{w,\Lambda_w} \leftarrow \mathsf{k}_{u,\Lambda_u} \oplus \mathsf{k}_{v,\Lambda_v}$. Observe that

$$\Lambda_w \leftarrow \Lambda_u \oplus \Lambda_v = (x \oplus \lambda_u) \oplus (y \oplus \lambda_v) = (x \oplus y) \oplus (\lambda_u \oplus \lambda_v) = (x \oplus y) \oplus \lambda_w$$

by the way the masks are constructed; similarly,

$$\mathsf{k}^j_{w,\Lambda_w} \leftarrow \mathsf{k}^j_{u,\Lambda_u} \oplus \mathsf{k}^j_{v,\Lambda_v} = (\mathsf{k}^j_{u,0} \oplus R^j \cdot \Lambda_u) \oplus (\mathsf{k}^j_{v,0} \oplus R^j \cdot \Lambda_v) = \mathsf{k}^j_{w,0} \oplus R^j \cdot (\Lambda_u \oplus \Lambda_v) = \mathsf{k}^j_{w,\Lambda_w}.$$

Output In the usual BMR protocol, immediately after garbling, the parties open the masks for output wires. This enables all parties to view the output bits. We will assume the parties simply hold the final output in secret-shared form, which they can do simply by not opening the output masks, and by computing (locally) the XOR of the public signal bit with the output mask. I.e. an output bit is shared as

$$[\![b]\!]_{2^k} \leftarrow [\![\lambda_w]\!]_{2^k} \oplus \Lambda_w.$$

In Section 3.2 we describe the modifications necessary to this standard garbling technique to provide inputs from get outputs to $\mathbb{F}_p$.

3 Protocol

In our protocol, one instance of $\mathcal{F}_{\mathsf{MPC}}$ over $\mathbb{F}_p$ is used to perform addition and multiplication in the field, and one instance of $\mathcal{F}_{\mathsf{MPC}}$ over $\mathbb{F}_{2^k}$ is used to perform the garbling. Note that since the keys for the PRF live in the field $\mathbb{F}_{2^k}$ in the garbling protocol, the instance of $\mathcal{F}_{\mathsf{MPC}}$ must be over a field with $k = O(\kappa)$ for computational security. Indeed, we emphasise that in our protocol k is not directly related to $\log p$. Once the garbling is completed, the full MPC engine in $\mathbb{F}_{2^k}$ is no longer required: the parties only maintain the $\mathbb{F}_p$ instance of $\mathcal{F}_{\mathsf{MPC}}$ and retain the garbled circuits in memory, and will additionally need to make sure they can still perform the procedure **Check** in $\mathcal{F}_{\mathsf{MPC}}$ on values opened in the evaluation of the GC.

In summary, our protocol requires a single opening of a secret-shared value and then locally bit-decomposing this public value to obtain the input wire signal bits to the garbled circuit. Once the parties have these, they open the appropriate keys for circuit evaluation, and the rest of the protocol (including retrieving outputs in secret-shared form) is local. The key challenge in creating the garbled circuit is that for some wire masks, namely a certain set of input masks and all the output wires, we need wire masks which are the same value in $\mathbb{F}_p$ and $\mathbb{F}_{2^k}$ (i.e. both the additive identity or both the multiplicative identity in each field), which then must be used in the garbling stage of the preprocessing.

We construct the functionality $\mathcal{F}_{\mathsf{CABB}}$ by first showing how to generate daBits, and then showing how this procedure coupled with two instances of the standard $\mathcal{F}_{\mathsf{MPC}}$ functionality gives a preprocessing phase which we call $\mathcal{F}_{\mathsf{Prep}}$, given in Figures 4 and 5, which can be used to realise $\mathcal{F}_{\mathsf{CABB}}$ via the SPDZ-BMR protocol with some slight modifications, called $\Pi_{\mathsf{ABB+BMR}}$. This is flow is outlined in Figure 3.

It would be straightforward to instantiate $\mathcal{F}_{\mathsf{CABB}}$ directly in the $\mathcal{F}_{\mathsf{MPC}}$-hybrid model, using two independent instances of $\mathcal{F}_{\mathsf{MPC}}$ over the fields $\mathbb{F}_p$ and $\mathbb{F}_{2^k}$. However, we choose to build up to $\mathcal{F}_{\mathsf{CABB}}$ via the functionality $\mathcal{F}_{\mathsf{Prep}}$ for three reasons:

1. This approach more faithfully resembles the execution of the protocols in our implementation, where daBits are generated and the "extended" $\mathcal{F}_{\mathsf{MPC}}$ functionality $\mathcal{F}_{\mathsf{Prep}}$ is used to run the extended BMR protocol.
2. The daBits are "raw" preprocessed data, so $\mathcal{F}_{\mathsf{Prep}}$ really forms a complete "offline" phase from which garbling and secret-sharing can be done.
3. Any future work giving a better protocol for creating daBits for two independent $\mathcal{F}_{\mathsf{MPC}}$ instances does not require reproving the security of the extended BMR protocol that makes use of daBits.

$$\begin{tikzcd} \mathcal{F}_{\mathsf{MPC}} \arrow[r, "\Pi_{\mathsf{daBits+MPC}}"] & \mathcal{F}_{\mathsf{Prep}} \arrow[r, "\Pi_{\mathsf{ABB+BMR}}"] & \mathcal{F}_{\mathsf{CABB}} \end{tikzcd}$$

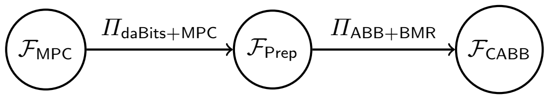

Fig. 3. Functionality dependencies

3.1 Generating daBits using Bucketing

Any technique for generating daBits require some form of checking procedure to ensure consistency between the two fields. Checking consistency often means checking random linear combinations of secrets produce the same result in both cases. Unfortunately, in our case such comparisons are meaningless since the fields have different characteristics, so shares are uniform in $\mathbb{F}_p$ and $\mathbb{F}_{2^k}$ and so multiplications in the field are not compatible. We can, however, check XORs of bits, which in $\mathbb{F}_p$ involves multiplication. (See Equation 1 in Section 2.) It is therefore necessary to use a protocol that minimises (as far as possible) the number of multiplications. Consequently, techniques using oblivious transfer (OT) such as [WRK17] to generated authenticated bits require a lot of XORs for checking correctness, so are undesirable for generating daBits.

Our chosen solution uses $\mathcal{F}_{\mathsf{MPC}}$ as a black box. In order to generate the same bit in both fields, each party samples a bit and calls the $\mathbb{F}_p$ and $\mathbb{F}_{2^k}$ instances of $\mathcal{F}_{\mathsf{MPC}}$ with this same input and then the parties compute the n-party XOR. To ensure all parties provided the same inputs in both fields, cut-and-choose and bucketing procedures are required, though since the number of bits it is necessary to generate is a multiple of $\log p \approx \mathsf{sec}$ and we can batch-produce daBits, the parameters are modest.

We use similar cut-and-choose and bucketing checks to those described by Frederiksen et al. [FKOS15, App. F.3], in which "triple-like" secrets can be efficiently checked. The idea behind these checks is the following. One first opens a random subset of secrets so that with some probability all unopened bits are correct. This ensures that the adversary cannot cheat on too many of the daBits. One then puts the secrets into buckets, and then in each bucket designates one secret as the one to output, uses all other secrets in bucket to check the last, and discards all but the designated secret. For a single bucket, the check will only pass (by construction) if either all secrets are correct or all are incorrect. Thus the adversary is forced to corrupt whole multiples of the bucket size and hope they are grouped together in the same bucket. Fortunately, (we will show that) there is no leakage on the bits since the parameters required for the parts of the protocol described above already preclude it. The protocol is described in Figure 6; we prove that this protocol securely realises the functionality $\mathcal{F}_{\mathsf{Prep}}$ in Figures 4 and 5 in the $\mathcal{F}_{\mathsf{MPC}}$-hybrid model. To do this, we require Proposition 1.

Proposition 1. *For a given $\ell > 0$, choose $B > 1$ and $C > 1$ so that $C^{-B} \cdot \binom{B\ell}{B}^{-1} < 2^{-\mathsf{sec}}$. Then the probability that one or more of the ℓ daBits output after **Consistency Check** by $\Pi_{\mathsf{daBits+MPC}}$ in Figure 6 is different in each field is at most $2^{-\mathsf{sec}}$.*

Proof. Using $\mathcal{F}^{p}_{\mathsf{MPC}}$ and $\mathcal{F}^{2^k}_{\mathsf{MPC}}$ as black boxes ensures the adversary can only possibly cheat in the input stage. We will argue that:

1. If both sets of inputs from corrupt parties to $\mathcal{F}^{p}_{\mathsf{MPC}}$ and $\mathcal{F}^{2^k}_{\mathsf{MPC}}$ are bits (rather than other field elements), then the bits are consistent in the two different fields with overwhelming probability.
2. The inputs in $\mathbb{F}_{2^k}$ are bits with overwhelming probability.
3. The inputs in $\mathbb{F}_p$ are bits with overwhelming probability.

We will conclude that the daBits are bits in the two fields, and are consistent.

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

This functionality extends the reactive functionality $\mathcal{F}_{\mathsf{MPC}}$ with commands to generate the same bits in two independent sessions.

<u>Instances of $\mathcal{F}_{\mathsf{MPC}}$</u>

Independent copies of $\mathcal{F}_{\mathsf{MPC}}$ are identified via session identifiers sid; $\mathcal{F}_{\mathsf{Prep}}$ maintains one dictionary $\mathsf{Val}_{\mathsf{sid}}$ for each instance. Entries cannot be changed, for simplicity. If a party provides an input with an sid which has not been initialised, output Reject to all parties and awaits another message.

Initialise On input $(\mathsf{Initialise}, \mathbb{F}, \mathsf{sid})$ from all parties, if sid is a new session identifer then initialise a database of secrets $\mathsf{Val}_{\mathsf{sid}}$ indexed by a set $\mathsf{Val}_{\mathsf{sid}}.\mathsf{Keys}$ and store the field as $\mathsf{Val}_{\mathsf{sid}}.\mathsf{Field} \leftarrow \mathbb{F}$. Set the internal flag $\texttt{Abort}_{\mathsf{sid}}$ to false.

Input On input $(\mathsf{Input}, i, \mathsf{id}, x, \mathsf{sid})$ from P_i and $(\mathsf{Input}, i, \mathsf{id}, \perp, \mathsf{sid})$ from all other parties, if $\mathsf{id} \notin \mathsf{Val}_{\mathsf{sid}}.\mathsf{Keys}$ then insert it and set $\mathsf{Val}_{\mathsf{sid}}[\mathsf{id}] \leftarrow x$. Then call the procedure **Wait**.

Add On input $(\mathsf{Add}, \mathsf{id}_x, \mathsf{id}_y, \mathsf{id}, \mathsf{sid})$, if $\mathsf{id}_x, \mathsf{id}_y \in \mathsf{Val}_{\mathsf{sid}}.\mathsf{Keys}$ then set $\mathsf{Val}_{\mathsf{sid}}[\mathsf{id}] \leftarrow \mathsf{Val}_{\mathsf{sid}}[\mathsf{id}_x] + \mathsf{Val}_{\mathsf{sid}}[\mathsf{id}_y]$.

Multiply On input $(\mathsf{Multiply}, \mathsf{id}_x, \mathsf{id}_y, \mathsf{id}, \mathsf{sid})$, if $\mathsf{id}_x, \mathsf{id}_y \in \mathsf{Val}_{\mathsf{sid}}.\mathsf{Keys}$ then set $\mathsf{Val}_{\mathsf{sid}}[\mathsf{id}] \leftarrow \mathsf{Val}_{\mathsf{sid}}[\mathsf{id}_x] \cdot \mathsf{Val}_{\mathsf{sid}}[\mathsf{id}_y]$. Then call the procedure **Wait**.

Random element On input $(\mathsf{RElt}, \mathsf{id}, \mathsf{sid})$, if $\mathsf{id} \notin \mathsf{Val}_{\mathsf{sid}}.\mathsf{Keys}$ then set $\mathsf{Val}_{\mathsf{sid}}[\mathsf{id}] \overset{\$}{\leftarrow} \mathsf{Val}_{\mathsf{sid}}.\mathsf{Field}$. Then call the procedure **Wait**.

Random bit On input $(\mathsf{RBit}, \mathsf{id}, \mathsf{sid})$, if $\mathsf{id} \notin \mathsf{Val}_{\mathsf{sid}}.\mathsf{Keys}$ then set $\mathsf{Val}_{\mathsf{sid}}[\mathsf{id}] \overset{\$}{\leftarrow} \{0, 1\}$. Then call the procedure **Wait**.

Open On input $(\mathsf{Open}, i, \mathsf{id}, \mathsf{sid})$ from all parties, if $\mathsf{id} \in \mathsf{Val}_{\mathsf{sid}}.\mathsf{Keys}$,
- if $i = 0$ then send $\mathsf{Val}_{\mathsf{sid}}[\mathsf{id}]$ to the adversary and run the procedure **Wait**. If the message was $(\mathsf{OK}, \mathsf{sid})$, await an error ε from the adversary. Send $\mathsf{Val}_{\mathsf{sid}}[\mathsf{id}] + \varepsilon$ to all honest parties and if $\varepsilon \neq 0$, set the internal flag $\texttt{Abort}_{\mathsf{sid}}$ to true.
- if $i \in A$, then send $\mathsf{Val}_{\mathsf{sid}}[\mathsf{id}]$ to the adversary and then run **Wait**.
- if $i \in [n] \setminus A$, then call the procedure **Wait**, and if not already halted then await an error ε from the adversary. Send $\mathsf{Val}_{\mathsf{sid}}[\mathsf{id}] + \varepsilon$ to P_i and if $\varepsilon \neq 0$ then set the internal flag $\texttt{Abort}_{\mathsf{sid}}$ to true.

Check On input $(\mathsf{Check}, \mathsf{sid})$ from all parties, run the procedure **Wait**. If not already halted and the internal flag $\texttt{Abort}_{\mathsf{sid}}$ is set to true, then send the message $(\mathsf{Abort}, \mathsf{sid})$ to the adversary and honest parties and ignore all further messages to $\mathcal{F}_{\mathsf{MPC}}$ with this sid; otherwise send the message $(\mathsf{OK}, \mathsf{sid})$ and continue.

Internal procedure:

Wait Await a message $(\mathsf{OK}, \mathsf{sid})$ or $(\mathsf{Abort}, \mathsf{sid})$ from the adversary; if the message is $(\mathsf{OK}, \mathsf{sid})$ then continue; otherwise, send the message $(\mathsf{Abort}, \mathsf{sid})$ to all honest parties and ignore all further messages to $\mathcal{F}_{\mathsf{MPC}}$ with this sid.

(continued...)

Fig. 4. Functionality $\mathcal{F}_{\mathsf{Prep}}$

Functionality $\mathcal{F}_{\mathsf{Prep}}$ (continued)

Additional commands

daBits On receiving $(\mathsf{daBits}, \mathsf{id}_1, \ldots, \mathsf{id}_\ell, \mathsf{sid}_1, \mathsf{sid}_2)$, from all parties where $\mathsf{id}_i \notin \mathsf{Val.Keys}$ for all $i \in \ell$, await a message OK or Abort from the adversary. If the message is OK, then sample $\{b_j\}_{j\in[\ell]} \xleftarrow{\$} \{0,1\}$ and for each $j \in [\ell]$, set $\mathsf{Val}_{\mathsf{sid}_1}[\mathsf{id}_j] \leftarrow b_j$ and $\mathsf{Val}_{\mathsf{sid}_2}[\mathsf{id}_j] \leftarrow b_j$ and insert the set $\{\mathsf{id}_i\}_{i\in[\ell]}$ into $\mathsf{Val}_{\mathsf{sid}_1}.\mathsf{Keys}$ and $\mathsf{Val}_{\mathsf{sid}_2}.\mathsf{Keys}$; otherwise send the messages $(\mathsf{Abort}, \mathsf{sid}_1)$ and $(\mathsf{Abort}, \mathsf{sid}_2)$ to all honest parties and the adversary and ignore all further messages to $\mathcal{F}_{\mathsf{MPC}}$ with session identifier sid_1 or sid_2.

Fig. 5. Functionality $\mathcal{F}_{\mathsf{Prep}}$ (continued)

1. Let c be the number of inconsistent daBits generated by a given corrupt party. If $c > B\ell$ then every set of size $(C-1)B\ell$ contains an incorrect daBit so the honest parties will always detect this in **Cut and Choose** and abort. Since $(C-1)B\ell$ out of $CB\ell$ daBits are opened, on average the probability that a daBit is not opened is $1-(C-1)/C = C^{-1}$, and so if $c < B\ell$ then we have:

$$\Pr[\text{None of the } c \text{ corrupted daBits is opened}] = C^{-c}. \tag{2}$$

At this point, if the protocol has not yet aborted, then there are $B\ell$ daBits remaining of which exactly c are corrupt.

Suppose a daBit $[\![b]\!]_{p,2^k}$ takes the value $\tilde{b}$ in $\mathbb{F}_p$ and $\hat{b}$ in $\mathbb{F}_{2^k}$. If the bucketing check passes then for every other daBit $[\![b']\!]_{p,2^k}$ in the bucket it holds that $\tilde{b} \oplus \tilde{b'} = \hat{b} \oplus \hat{b'}$, so $\tilde{b'} = (\hat{b} \oplus \hat{b'}) \oplus \tilde{b}$, and so $\tilde{b} = \hat{b} \oplus 1$ if and only if $\tilde{b'} = \hat{b'} \oplus 1$. (Recall that we are assuming the inputs are certainly bits at this stage.) In other words, within a single bucket, the check passes if and only if either all daBits are inconsistent, or if none of them are. Thus the probability **Consistency Check** passes without aborting is the probability that all corrupted daBits are placed into the same buckets. Moreover, this implies that if the number of corrupted daBits, c, is not a multiple of the bucket size, this stage never passes, so we write $c = Bt$ for some $t > 0$. Then we have:

$$\begin{aligned}
&\Pr[\text{All corrupted daBits are placed in the same buckets}] = \\
&\frac{\binom{Bt}{B}\cdot\binom{B(t-1)}{B}\cdots\binom{B}{B}\cdot\binom{B\ell-Bt}{B}\cdot\binom{B\ell-Bt-B}{B}\cdots\binom{B}{B}}{\binom{B\ell}{B}\cdot\binom{B\ell-B}{B}\cdots\binom{B}{B}} \\
&= \frac{(Bt)!}{B!^t}\cdot\frac{(B\ell-Bt)!}{B!^{\ell-t}}\cdot\frac{B!^\ell}{(B\ell)!} = \binom{B\ell}{Bt}^{-1}.
\end{aligned} \tag{3}$$

Since the randomness for **Cut and Choose** and **Check Correctness** is independent, the event that both checks pass after the adversary corrupts c daBits is the product of the probabilities. To upper-bound the adversary's chance of winning, we compute the probability by maximising over t: thus we need C and B so that

$$\max_t \left\{ C^{-Bt} \cdot \binom{B\ell}{Bt}^{-1} \right\} < 2^{-\mathsf{sec}} \tag{4}$$

Protocol $\Pi_{\mathsf{daBits+MPC}}$

This protocol is in the $\mathcal{F}_{\mathsf{MPC}}$-hybrid model.

Initialise

1. Call an instance of $\mathcal{F}_{\mathsf{MPC}}$ with input $(\mathsf{Initialise}, \mathbb{F}_p, 0)$; denote it by $\mathcal{F}^p_{\mathsf{MPC}}$.
2. Call an instance of $\mathcal{F}_{\mathsf{MPC}}$ with input $(\mathsf{Initialise}, \mathbb{F}_{2^k}, 1)$; denote it by $\mathcal{F}^{2^k}_{\mathsf{MPC}}$.

Calls to $\mathcal{F}_{\mathsf{MPC}}$ Dealt with by $\mathcal{F}^p_{\mathsf{MPC}}$ or $\mathcal{F}^{2^k}_{\mathsf{MPC}}$, as appropriate.

To generate ℓ bits, all of the following procedures are performed, in order.

Generate daBits

1. Let $m \leftarrow CB\ell$ where $C > 1$ and $B > 1$ are chosen so that $C^B \cdot \binom{B\ell}{B} > 2^{\mathsf{sec}}$.
2. For each $i \in [n]$,
   (a) Party P_i samples a bit string $(b^i_1, \dots, b^i_m) \xleftarrow{\$} \{0,1\}^m$.
   (b) Call $\mathcal{F}^p_{\mathsf{MPC}}$ where P_i has input $(\mathsf{Input}, i, [\![b^i_j]\!]_p, b^i_1, 0)^m_{j=1}$ and P_j $(j \neq i)$ has input $(\mathsf{Input}, i, [\![b^i_j]\!]_p, \perp, 0)^m_{i=1}$.
   (c) Call $\mathcal{F}^{2^k}_{\mathsf{MPC}}$ where P_i has input $(\mathsf{Input}, i, [\![b^i_j]\!]_{2^k}, b^i_1, 1)^m_{j=1}$ and P_j $(j \neq i)$ has input $(\mathsf{Input}, i, [\![b^i_j]\!]_{2^k}, \perp, 1)^m_{i=1}$.

Cut and Choose

1. Call $\mathcal{F}_{\mathsf{Rand}}$ with input $(\mathsf{RSubset}, [CB\ell], (C-1)B\ell)$ to obtain a set S.
2. Call $\mathcal{F}^p_{\mathsf{MPC}}$ with inputs $(\mathsf{Open}, 0, [\![b^i_j]\!]_p, 0)_{j \in S}$ for all $i \in [n]$.
3. Call $\mathcal{F}^{2^k}_{\mathsf{MPC}}$ with inputs $(\mathsf{Open}, 0, [\![b^i_j]\!]_{2^k}, 1)_{j \in S}$ for all $i \in [n]$.
4. If any party sees daBits which are not in $\{0,1\}$ or not the same in both fields, they send the message Abort to all parties and halt.

Combine For all $j \in S$, do the following:

1. Set $[\![b_j]\!]_p \leftarrow [\![b^1_j]\!]_p$ and then for i from 2 to n compute
   (a) $[\![b_j]\!]_p \leftarrow [\![b_j]\!]_p + [\![b^i_j]\!]_p - 2 \cdot [\![b_j]\!]_p \cdot [\![b^i_j]\!]_p$
2. Compute $[\![b_j]\!]_{2^k} \leftarrow \bigoplus^n_{i=1} [\![b^i_j]\!]_{2^k}$.

Consistency Check

1. Call $\mathcal{F}_{\mathsf{Rand}}$ with input $(\mathsf{RBucket}, [B\ell], B)$ and use the returned sets $(S_i)^\ell_{i=1}$ to put the $B\ell$ daBits into ℓ buckets of size B.
2. For each bucket S_i,
   (a) Relabel the bits in this bucket as $b^1, \dots, b^B$.
   (b) For $j = 2$ to B, compute $[\![c^j]\!]_p \leftarrow [\![b^1]\!]_p + [\![b^j]\!]_p - 2 \cdot [\![b^1]\!]_p \cdot [\![b^j]\!]_p$ and $[\![c^j]\!]_{2^k} \leftarrow [\![b^1]\!]_{2^k} \oplus [\![b^j]\!]_{2^k}$.
   (c) Call $\mathcal{F}^p_{\mathsf{MPC}}$ with inputs $(\mathsf{Open}, 0, [\![c^j]\!]_p, 0)^B_{j=2}$.
   (d) Call $\mathcal{F}^{2^k}_{\mathsf{MPC}}$ with inputs $(\mathsf{Open}, 0, [\![c^j]\!]_{2^k}, 1)^B_{j=2}$.
   (e) If any party sees daBits which are not in $\{0,1\}$ or not the same in both fields, they send the message Abort to all parties and halt.
   (f) Set $[\![b_i]\!]_{p,2^k} \leftarrow [\![b^1]\!]_{p,2^k}$.
3. Call $\mathcal{F}^p_{\mathsf{MPC}}$ with input $(\mathsf{Check}, 0)$.
4. Call $\mathcal{F}^{2^k}_{\mathsf{MPC}}$ with input $(\mathsf{Check}, 1)$.
5. If the checks pass without aborting, output $\{[\![b_i]\!]_{p,2^k}\}^\ell_{i=1}$ and discard all other bits.

Fig. 6. Protocol $\Pi_{\mathsf{daBits+MPC}}$

The maximum occurs when t is small, and $t \geq 1$ otherwise no cheating occurred; thus since the proposition stipulates that $C^{-B} \cdot \binom{B\ell}{B}^{-1} < 2^{-\mathsf{sec}}$, the daBits are consistent in both fields, if they are indeed bits in both fields.

2. Next, we will argue that the check in **Cut and Choose** ensures that the inputs given to $\mathcal{F}_{\mathsf{MPC}}^{2^k}$ are indeed bits. It follows from Equation 2 that the step **Cut and Choose** aborts with probability C^{-c} if any element of either field is not a bit, as well as if the element in the two fields does not match. Moreover, in **Consistency Check**, in order for the check to pass in $\mathbb{F}_{2^k}$ for a given bucket, the secrets' higher-order bits must be the same for all shares so that the XOR is always zero when the pairwise XORs are opened. Thus the probability that this happens is the same as the probability above in Equation 4 since again this can only happen when the adversary is not detected in **Cut and Choose**, that he cheats in some multiple of B daBits, and that these cheating bits are placed in the same buckets in **Consistency Check**.

3. We now show that all of the the $\mathbb{F}_p$ components are bits. To do this, we will show that if the $\mathbb{F}_p$ component of a daBit is not a bit, then the bucket check passes only if all other daBits in the bucket are also not bits in $\mathbb{F}_p$.

If the protocol has not aborted, then in every bucket B, for every $2 \leq j \leq B$, it holds that

$$b^1 + b^j - 2 \cdot b^1 \cdot b^j = c^j \tag{5}$$

where $c^j \in \{0, 1\}$ are determined by the XOR in $\mathbb{F}_{2^k}$. Note that since $c^j = \bigoplus_{i=1}^n b_i^1 \oplus \bigoplus_{i=1}^n b_i^j$ and at least one b_i^j is generated by an honest party, this value is uniform and unknown to the adversary when he chooses his inputs at the beginning.

Suppose $b^1 \in \mathbb{F}_p \setminus \{0, 1\}$. If $b^1 = 2^{-1} \in \mathbb{F}_p$ then by Equation 5 we have $b^1 = c^j$; but c^j is a bit, so the "XOR" is not the same in both fields and the protocol will abort. Thus we may assume $b^1 \neq 2^{-1}$ and so we can rewrite the equation above as

$$b^j = \frac{b^1 - c^j}{2 \cdot b^1 - 1}. \tag{6}$$

Now if b^j is a bit then it satisfies $b^j(b^j - 1) = 0$, and so

$$0 = \left(\frac{b^1 - c^j}{2 \cdot b^1 - 1}\right) \cdot \left(\frac{b^1 - c^j}{2 \cdot b^1 - 1} - 1\right) = -\frac{(b^1 - c^j)(b^1 - (1 - c^j))}{(2 \cdot b^1 - 1)^2}$$

so $b^1 = c^j$ or $b^1 = 1 - c^j$; thus $b^1 \in \{0, 1\}$, which is a contradiction. Thus we have shown that if b^1 is not a bit then b^j is not a bit for every other b^j in this bucket. Moreover, for each $j = 2, \ldots, B$, there are two distinct values $b^j \in \mathbb{F}_p \setminus \{0, 1\}$ solving Equation 6 corresponding to the two possible values of $c^j \in \{0, 1\}$, which means that if the bucket check passes then the adversary must *also* have guessed the bits $\{c^j\}_{j=1}^B$, which he can do with probability 2^{-B} since they are constructed using at least one honest party's input. Thus the chance of cheating without detection in this way is at most $2^{-Bt} \cdot C^{-Bt} \cdot \binom{B\ell}{Bt}^{-1}$.

Thus we have shown that the probability that $b^1 \in \mathbb{F}_p \setminus \{0, 1\}$ is given as output for the $\mathbb{F}_p$ component is at most the probability that the adversary corrupts a multiple of B daBits, that these daBits are placed in the same buckets, and that the adversary correctly guesses c bits from honest parties (in the construction of the bits $\{b^j\}_{j \in B}$) so that the appropriate equations hold in the corrupted buckets. Indeed, needing to guess the bits ahead of time only *reduces* the adversary's chance of winning from the same probability in the $\mathbb{F}_{2^k}$ case.

We conclude that the daBits are bits in both fields and are the same in both fields with probability except with probability at most $2^{-\mathsf{sec}}$. $\square$

Theorem 1. *The protocol $\Pi_{\mathsf{daBits+MPC}}$ securely realises $\mathcal{F}_{\mathsf{Prep}}$ in the $(\mathcal{F}_{\mathsf{MPC}}, \mathcal{F}_{\mathsf{Rand}})$-hybrid model against an active adversary corrupting up to $n-1$ out of n parties.*

Proof. To prove security in the UC framework we must show that to any environment $\mathcal{Z}$, for any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that the execution of an idealised version of the protocol run by a trusted third party $\mathcal{F}$ with the simulator is indistinguishable from a real execution of the protocol Π between the honest parties and the adversary. The environment specifies the code run by the adversary as well as the inputs of all parties, honest and dishonest. Additionally, the environment sees all outputs of all parties; it does not see the intermediate interactions in subroutines of the honest parties' executions, otherwise distinguishing would be trivial as honest parties either perform Π or interact with $\mathcal{F}$. In the $(\mathcal{F}_{\mathsf{MPC}}, \mathcal{F}_{\mathsf{Rand}})$-hybrid model, the adversary is allowed to make oracle queries to these functionalities and $\mathcal{S}$ must generate the responses.

Note the functionality does *not* have access to the random tapes honest parties as this would make distinguishing between worlds trivial: it would be impossible for the simulator to emulate honest parties to the real-world adversary indistinguishably since for any random tape sampled by the simulator, the environment would always be able to execute the protocol internally, using its knowledge of the random tapes of honest parties to execute the entire protocol deterministically, and compare it to the output of the simulator.

Following standard practice, and as described in [Can00, §4.2.2], we define a simulator which interacts with the adversary $\mathcal{A}$ as a black box. This allows us to make the claim that the simulator works regardless of the code run by the adversary and hence prove the claim.

Suppose the adversary corrupts $t < n$ parties in total, indexed by a set A. We define a sequence of hybrid worlds $(\textbf{Hybrid } h)_{h=0}^{n-t}$ and show that each is indistinguishable from the previous. **Hybrid** h is defined as:

Hybrid h The simulator has the actual input of $n-t-h$ honest parties and must simulate the remaining h honest parties towards the adversary.

The simulator is described in Figure 7.

Claim. The $\mathcal{F}_{\mathsf{MPC}}, \mathcal{F}_{\mathsf{Rand}}$-hybrid world is indistinguishable from **Hybrid** 0.

Proof. Correctness of the simulation holds as follows. The simulator emulates $\mathcal{F}_{\mathsf{MPC}}^{p}$, $\mathcal{F}_{\mathsf{MPC}}^{2^k}$ and $\mathcal{F}_{\mathsf{Rand}}$, so all calls made to these oracles are dealt with as in an execution of the protocol. Indeed, for all calls to $\mathcal{F}_{\mathsf{MPC}}$ in either field which are outside of the daBits generation procedure, the commands are forwarded to $\mathcal{F}_{\mathsf{Prep}}$ and relayed back to $\mathcal{A}$, and since $\mathcal{F}_{\mathsf{Prep}}$ has the same interface as $\mathcal{F}_{\mathsf{MPC}}$ by definition, there is no difference between the worlds. As for the daBit generation, when the adversary makes calls to provide (random) inputs and then perform **Cut and Choose**, the simulator does not forward the messages through to $\mathcal{F}_{\mathsf{Prep}}$ since all bits used in the protocol except the final output bits are discarded. Instead the command $(\mathsf{daBits}, \mathsf{id}_1, \dots, \mathsf{id}_\ell, 0, 1)$ is sent to $\mathcal{F}_{\mathsf{Prep}}$ and the simulator executes the daBit routines honestly with the adversary, making random choices for honest parties by sampling in the same way as in the protocol.

Now we argue indistinguishability between executions: we must show that for any algorithm $\mathcal{A}$ specified by the environment $\mathcal{Z}$, it holds that

$$\textsc{Exec}(\mathcal{Z}, \mathcal{A}^{\mathcal{F}_{\mathsf{MPC}}, \mathcal{F}_{\mathsf{Rand}}}, \Pi_{\mathsf{daBits+MPC}}) \sim \textsc{Exec}(\mathcal{Z}, \mathcal{S}_{\mathsf{Prep}}^{0}, \mathcal{F}_{\mathsf{Prep}})$$

where $\sim$ denotes statistical indistinguishability of distributions, and the randomness of these distributions is taken over the random tapes of honest parties and the adversary and simulator.

First, note that the oracles $\mathcal{F}_{\mathsf{MPC}}$ and $\mathcal{F}_{\mathsf{Rand}}$ are executed honestly by $\mathcal{S}_{\mathsf{Prep}}^{0}$ so the contribution to the distributions is the same in both executions.

Simulator $\mathcal{S}^h_{\mathsf{Prep}}$

The simulator is (vacuously) parameterised by h, which means the simulator knows the actual inputs of $n-t-h$ honest parties, and must simulate for the remaining h. We denote the adversary by $\mathcal{A}$.

Initialise On receiving the call to $\mathcal{F}_{\mathsf{MPC}}$ with inputs $(\mathsf{Initialise}, \mathbb{F}_p, 0)$ and $(\mathsf{Initialise}, \mathbb{F}_{2^k}, 1)$, initialise corresponding internal copies.

Calls to $\mathcal{F}^p_{\mathsf{MPC}}$ All calls for producing preprocessing, other than what is described below, sent from $\mathcal{A}$ to $\mathcal{F}^p_{\mathsf{MPC}}$ should be forwarded to $\mathcal{F}_{\mathsf{Prep}}$. All response messages from $\mathcal{F}_{\mathsf{Prep}}$ are sent directly to $\mathcal{A}$.

Calls to $\mathcal{F}^{2^k}_{\mathsf{MPC}}$ All calls for producing preprocessing, other than what is described below, sent from the $\mathcal{A}$ to $\mathcal{F}^{2^k}_{\mathsf{MPC}}$ should be forwarded to $\mathcal{F}_{\mathsf{Prep}}$. All response messages from $\mathcal{F}_{\mathsf{Prep}}$ are sent directly to $\mathcal{A}$.

For the following procedures, send the calls to the *internal* copies of $\mathcal{F}^p_{\mathsf{MPC}}$, $\mathcal{F}^{2^k}_{\mathsf{MPC}}$ and $\mathcal{F}_{\mathsf{Rand}}$ as described in the protocol.

[Start] Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{daBits}, \mathsf{id}_1, \ldots, \mathsf{id}_\ell, 0, 1)$.

Generate daBits Run **Generate daBits** from $\Pi_{\mathsf{daBits+MPC}}$ with $\mathcal{A}$, sampling inputs for all honest parties.

Cut and Choose Run **Cut and Choose** from $\Pi_{\mathsf{daBits+MPC}}$ with $\mathcal{A}$.

Combine Run **Combine** from $\Pi_{\mathsf{daBits+MPC}}$ with $\mathcal{A}$.

Check Correctness Run **Check Correctness** from $\Pi_{\mathsf{daBits+MPC}}$ with $\mathcal{A}$.

[Finish] If the protocol aborted, send Abort to $\mathcal{F}_{\mathsf{Prep}}$, and otherwise send OK.

Fig. 7. Simulator $\mathcal{S}^h_{\mathsf{Prep}}$

Second, since the inputs of honest parties are sampled during the protocol, they are not specified or known by the environment. However, if the adversary performs a *selective-failure attack*, then the environment may learn information. A selective failure attack is where the environment can learn some information if the protocol does not detect cheating behaviour. For example, if the environment guesses an entire bucket of bits and chooses inputs for the adversary's input so that the bucket check would pass based on these guesses, then if the protocol does not abort then the environment learns that its guesses were correct. Then if the final output bit is *not* the XOR of all parties' inputs then the execution must have happened in **Hybrid** 0 since in this world the output depends on the random tape of $\mathcal{F}_{\mathsf{Prep}}$ and is independent of the adversary's and honest parties' random tapes, contrasting the output in the $\mathcal{F}_{\mathsf{MPC}}, \mathcal{F}_{\mathsf{Rand}}$-hybrid world in which the final output is an XOR of bits on these tapes (which were guessed by the environment). Since this happens with probability $\frac{1}{2}$, in expected 2 executions, the environment can distinguish. However, by Proposition 1, the environment can only mount a selective failure attack with success with probability at most $2^{-\mathsf{sec}}$ by the choice of parameters.

Thus the only way to distinguish between worlds is if the transcript leaks information on the honest parties' inputs. In **Check Correctness**, XORs are computed in both fields and the result is opened; however, this reveals no information on the final daBit outputs as the linear dependence between the secret and the public values is broken by discarding all secrets in each bucket except the designated (i.e. first) bit. We conclude that the overall distributions of the two executions are statistically indistinguishable in sec. ■

Claim. **Hybrid** h is indistinguishable from **Hybrid** $h+1$ for $h = 0, \dots, n-t-1$.

Proof. There is no difference between these worlds since honest parties' (random) inputs are sampled the same way in both cases. ■

Since $\mathcal{F}_{\mathsf{MPC}}$ is secure up to $t = n-1$, the result follows. □

3.2 Garbling and Switching

In this section we describe the garbling, and how the parties provide input to the circuit from secret-shared data, and *vice versa*. In Figure 8 we show pictographically what happens at the barrier between secret-sharing and garbling, though note we have shown a circuit output that will be reconstructed to an element of $\mathbb{F}_p$: the circuit output can be any string of bits. We first discuss the methods of switching into and out of the garbled circuit, and then present the modified SPDZ-BMR protocol $\Pi_{\mathsf{ABB+BMR}}$ in Figures 10, 11, 12 and 13 which we show securely realises $\mathcal{F}_{\mathsf{CABB}}$ in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model.

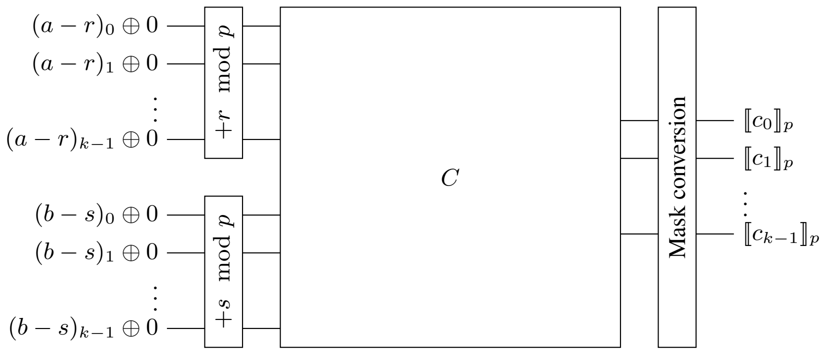

Fig. 8. Overview

Correctness of providing circuit inputs In brief, the parties open $a-r$ in MPC where $r = \sum_{j=0}^{\lfloor \log p \rfloor -1} 2^j \cdot [\![r_j]\!]_p$ is constructed from daBits $\{[\![r_j]\!]_{p,2^k}\}_{j=0}^{\lfloor \log p \rfloor -1}$, and $\mathcal{F}_{\mathsf{MPC}}$ is called with input $(\mathsf{Check}, 0)$ either at this point or later on, and then these public values are taken to be signal bits to the circuit. To correct the offset r, the circuit simply takes in the $\mathbb{F}_{2^k}$ parts of the daBits of r as input, and the circuit $(a-r)+r \mod p$ is computed inside the garbled circuit. Since $a-r$ is a public value, there is no need to have masks for the corresponding input wires, so they are set to 0. Furthermore, since r is independent of the online inputs, the keys corresponding to these inputs can be obtained directly during garbling in Step 6 of **Input layer** instead of waiting to open keys in the online phase.

In more detail, we define the following Boolean circuit

$$\mathtt{ADDMOD}(x, y, p) \leftarrow (x+y) - p \cdot \left((x+y) \overset{?}{\geq} p \right)$$

where the computation takes place over the integers and the inputs x and y are supplied as a string of bits.

Let the input wires of $\texttt{ADDMOD}(x, y, p)$ be $(u_j)_{j=0}^{\lfloor \log p \rfloor - 1}$ for the bits of x, $(v_j)_{j=0}^{\lfloor \log p \rfloor - 1}$ for the bits of y, and the output wires be $(w_j)_{j=0}^{\lfloor \log p \rfloor - 1}$, and let the input wires of C be $(u'_j)_{j=0}^{\lfloor \log p \rfloor - 1}$. Then the circuit that the parties garble in $\Pi_{\mathsf{ABB+BMR}}$ is the circuit obtained by associating wire w_j with wire u'_j for all $j = 0$ to $\lfloor \log p \rfloor - 1$. Now if $x = a - r$ and $y = r$ then clearly $C \circ \texttt{ADDMOD}(x, y, p) = C(a)$ where $\circ$ denotes the wiring association as above.

An obvious optimisation that we use in our implementation is to note that if the masking bits $[\![\lambda_{w_{i,j}}]\!]_{2^k}$ for inputs $y_{i,j}$ are just taken to be $[\![r_{i,j}]\!]_{2^k}$ then the signal bits are 0 and this reveals nothing about $r_{i,j}$, so we save on generating $\log p$ masking bits.

Correctness of receiving circuit outputs The output of a multiparty garbled circuit is one or more keys and corresponding public signal bits. In normal SPDZ-BMR, the output wire masks are revealed after garbling so that all parties can learn the final outputs. A simple way of retaining shared output of the circuit, which is what we want, is for the parties not to reveal the masks for output wires after garbling and instead to compute the XOR of the secret-shared mask with the public signal bit, in MPC. In other words, for output wire w they obtain a sharing of the secret output bit b by computing

$$[\![b]\!]_{2^k} \leftarrow \Lambda_w \oplus [\![\lambda_w]\!]_{2^k}.$$

In our case, we want the shared output of the circuit to be in $\mathbb{F}_p$, and to do this it suffices for the masks on circuit output wires to be daBits and for the parties to compute (locally)

$$[\![b]\!]_p \leftarrow \Lambda_w + [\![\lambda_w]\!]_p - 2 \cdot \Lambda_w \cdot [\![\lambda_w]\!]_p.$$

To make our whole approach more more modular, we define an additional layer to the circuit after the output layer which converts output wires with masks only in $\mathbb{F}_{2^k}$ to output wires with masks as daBits, without changing the real values on the wire. To do this, parties do the following: for every output wire w,

1. In the garbling stage,
   (a) Take a new daBit $[\![\lambda_{w'}]\!]_{p,2^k}$.
   (b) Set $[\![\lambda_{w_0}]\!]_{2^k} \leftarrow [\![\lambda_w]\!]_{2^k} \oplus [\![\lambda_{w'}]\!]_{2^k}$
   (c) Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{Open}, 0, 0 \oplus [\![\lambda_{w_0}]\!]_{2^k})$ an call this Λ_{w_0}.
2. In the evaluation stage, upon obtaining Λ_w,
   (a) Compute $\Lambda_{w'} \leftarrow \Lambda_w \oplus \Lambda_{w_0}$.
   (b) Compute the final ($\mathbb{F}_p$-secret-shared) output as $[\![b]\!]_p \leftarrow \Lambda_{w'} + [\![\lambda_{w'}]\!]_p - 2 \cdot \Lambda_{w'} \cdot [\![\lambda_{w'}]\!]_p$.

Observe that $\Lambda_{w_0} \equiv \lambda_{w_0}$ so this procedure is just adding a layer of XOR gates where the masking bits are daBits and the other input wire is always 0 (so the gate evaluation doesn't change the real wire value). Note that since the signal bits for XOR gates are determined from input signal bits and not the output key, there is no need to generate an output key for wire w_0. For correctness, observe that:

$$\begin{aligned}\Lambda_{w'} \oplus \lambda_{w'} &= (\Lambda_w \oplus \Lambda_{w_0}) \oplus (\lambda_w \oplus \lambda_{w_0}) \\ &= ((b \oplus \lambda_w) \oplus (0 \oplus \lambda_{w_0})) \oplus (\lambda_w \oplus \lambda_{w_0}) \\ &= b.\end{aligned}$$

Correctness of the actual garbling was outlined in Section 2. The proof of Theorem 2 is deferred to the appendices as it is straightforward, following from the security of SPDZ-BMR [LPSY15] and the fact that the additional input and output procedures perfectly hide the actual circuit inputs and outputs.

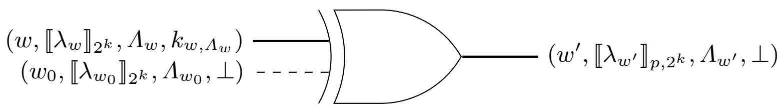
$(w, [\![\lambda_w]\!]_{2^k}, \Lambda_w, k_{w,\Lambda_w})$
$(w_0, [\![\lambda_{w_0}]\!]_{2^k}, \Lambda_{w_0}, \perp)$
$(w', [\![\lambda_{w'}]\!]_{p,2^k}, \Lambda_{w'}, \perp)$

Fig. 9. Circuit output wires

Theorem 2. *$\Pi_{\mathsf{ABB+BMR}}$ securely realises $\mathcal{F}_{\mathsf{CABB}}$ in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model.*

Proof. See Appendix C.

4 Implementation

We have implemented daBit generation and the conversion between arithmetic shares and garbled circuits. Our code is developed on top of the MP-SPDZ framework [Ana19] and experiments were run on computers with commodity hardware connected via a 1Gb/s LAN connection with an average round-trip ping time of 0.3ms. The $\mathcal{F}^p_{\mathsf{MPC}}$ functionality is implemented using LowGear, one of the two variants of Overdrive [KPR18]; the $\mathcal{F}^{2^k}_{\mathsf{MPC}}$ functionality is implemented using MASCOT [KOS16]. In our experiments, $\mathbb{F}_{2^k}$ is always taken with $k = \kappa = 128$ since this is the security of PRF keys used in SPDZ-BMR. The daBits are always generated with $\kappa = 128$ and the same statistical security sec as the protocol for $\mathcal{F}_{\mathsf{MPC}}$.

Primes. We require that p be close to a power of 2 so that $a - r$ is indistinguishable from a uniform element of the field, as discussed in Section 2. Since we use LowGear in our implementation, for a technical reason we also require that p be congruent to $1 \mod N$ where $N = 32768$. (This is the amount of packing in the ciphertexts.) Consequently, using LowGear means we always lose $15 = \log 32768$ bits of security if $p > 65537$ since then the k-bit prime must be of the form $2^{k-1} + t \cdot 2^{15} + 1$ for some t where $1 \leq t \leq 2^{k-16} - 1$, so the secret masks r constructed from a sequence of bits "miss" at least this much of the field.

Cut and choose optimisation. One key observation that enables reduction of the preprocessing overhead in $\mathbb{F}_{2^k}$ is that parties only need to input bits (instead of full $\mathbb{F}_{2^k}$ field elements) into $\mathcal{F}_{\mathsf{MPC}}$ during $\Pi_{\mathsf{daBits+MPC}}$. For a party to input a secret x in MASCOT, the parties create a random authenticated mask r and open opened it to the party, and the party then broadcasts $x + r$. Since the inputs are just bits, it suffices for the random masks also to be bits. Generating authenticated bits using MASCOT is extremely cheap and comes with a small communication overhead (see Table 2).

More efficient packing for MAC Check. Instead of a set of k secret bits being opened as full $\mathbb{F}_{2^k}$ field elements $(0, \ldots, 0, b_1), \ldots, (0, \ldots, 0, b_t) \in \mathbb{F}_2^k \cong \mathbb{F}_{2^k}$, we can save on all the redundant 0's being sent by sending a single field element $(b_k, \ldots, b_1) \in \mathbb{F}_{2^k}$. This optimisation reduces by a factor 2 the amount of data sent for the online phase of daBit generation.

Complexity analysis. In LowGear (Overdrive) and MASCOT the authors choose to avoid reporting any benchmarks for random bit masks in $\mathbb{F}_{2^k}$ or random input masks in $\mathbb{F}_p$ since they focused on the entire triple generation protocol. Fortunately their code is open source and easy to modify so we micro-benchmarked their protocols in order to get concrete costs for the procedure **Input** for $\mathcal{F}^p_{\mathsf{MPC}}$ and $\mathcal{F}^{2^k}_{\mathsf{MPC}}$. For example, in the two-party case, to provide an input bit costs overall 0.384kbits with MASCOT in $\mathbb{F}_{2^k}$. For LowGear

Protocol $\Pi_{\mathsf{ABB+BMR}}$

This protocol is secure in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model.

Initialise The parties call $\mathcal{F}_{\mathsf{Prep}}$ with inputs $(\mathsf{Initialise}, \mathbb{F}_p, 0)$ and $(\mathsf{Initialise}, \mathbb{F}_{2^k}, 1)$.

Arithmetic

Input For P_i to provide input $x \in \mathbb{F}_p$, P_i calls $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{Input}, i, [\![x]\!]_p, x, 0)$ and all other parties call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{Input}, P_i, [\![x]\!]_p, \perp, 0)$, where $[\![x]\!]_p$ is a fresh identifier.

Add To add secrets x and y, parties call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{Add}, [\![x]\!]_p, [\![y]\!]_p, [\![z]\!]_p, 0)$ where $[\![z]\!]_p$ is a new identifier.

Multiply To multiply secrets x and y, parties call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{Add}, [\![x]\!]_p, [\![y]\!]_p, [\![z]\!]_p, 0)$ where $[\![z]\!]_p$ is a new identifier.

Output To receive output with identifier id, parties do the following:
1. The parties call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{Check}, 0)$.
2. The parties call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{Open}, 0, \mathsf{id}, 0)$.
3. The parties call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{Check}, 0)$.

Circuit (All of the following procedures are performed, in order.)

Initialise garbling To garble a Boolean circuit C with identifiers W for wires, G_{AND} for AND gates and G_{XOR} for XOR gates, the parties do the following:
1. The parties call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{daBits}, \{[\![\lambda_w]\!]_{p,2^k}\}_{w \in W_o}, 0, 1)$ where W_o denotes the set indexing circuit output wires.
2. For each $i \in [n]$, the parties call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{RElt}, [\![R^i]\!]_{2^k}, 1)$ and then call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{Open}, i, [\![R^i]\!]_{2^k}, 1)$ to reveal R^i to P_i.

Input layer Let the number of $\mathbb{F}_p$ inputs to the circuit be t. The parties do the following:
1. Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{daBits}, (\{[\![r_{i,j}]\!]_{p,2^k}\}_{j=0}^{\lfloor \log p \rfloor -1})_{i=1}^{t}, 0, 1)$.
2. Set $[\![r_i]\!]_p \leftarrow \sum_{j=0}^{\lfloor \log p \rfloor -1} 2^j [\![r_{i,j}]\!]_p$.
3. For $i = 1, \dots, t$, create the circuit $\mathtt{ADDMOD}(x_i, y_i, p)$ and prepend these circuits to the circuit C to be garbled, augmenting G_{AND} and G_{XOR} as appropriate. See Section 3.2 for details.
4. For each input wire $w \in W$, for each $i \in [n]$,
   (a) Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{RElt}, [\![\mathsf{k}^i_{w,0}]\!]_{2^k}, 1)$.
   (b) Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{Open}, i, [\![\mathsf{k}^i_{w,0}]\!]_{2^k}, 1)$ to reveal $\mathsf{k}^i_{w,0}$ to P_i.
   (c) P_i sets the one key as $\mathsf{k}^i_{w,1} \leftarrow \mathsf{k}^i_{w,0} \oplus R^i$ and the parties set $[\![\mathsf{k}^i_{w,1}]\!]_{2^k} \leftarrow [\![\mathsf{k}^i_{w,0}]\!]_{2^k} \oplus [\![R^i]\!]_{2^k}$.
5. For every input wire w corresponding to an input $x_{i,j}$ of $\mathtt{ADDMOD}(x_i, y_i, p)$, set $\lambda_{w_{i,j}} \leftarrow 0$.
6. For every input wire w corresponding to an input $y_{i,j}$ of $\mathtt{ADDMOD}(x_i, y_i, p)$, call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{RBit}, [\![\lambda_w]\!]_{2^k}, 1)$ followed by $(\mathsf{Open}, 0, [\![r_{i,j}]\!]_{2^k} \oplus [\![\lambda_{w_{i,j}}]\!]_{2^k}, 1)$ and store this as $\Lambda_{w_{i,j}}$. Then for every $l \in [n]$, P_l sends $k^l_{w_{i,j}, \Lambda_{w_{i,j}}}$ to all other parties.

Garble Refer to $\Pi^{\mathsf{Garble}}_{\mathsf{ABB+BMR}}$ in Figure 12.

(continued...)

Fig. 10. Protocol $\Pi_{\mathsf{ABB+BMR}}$

Protocol $\Pi_{\mathsf{ABB+BMR}}$ (continued)

Output layer For every wire w that is an (external, circuit) output wire, the parties do the following

1. Retrieve a daBit $[\![\lambda_{w'}]\!]_{p,2^k}$ from memory, generated in **Initialise**.
2. Compute $[\![\lambda_{w_0}]\!]_{2^k} \leftarrow [\![\lambda_w]\!]_{2^k} \oplus [\![\lambda_{w'}]\!]_{2^k}$.
3. Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{Open}, 0, [\![\lambda_{w_0}]\!]_{2^k}, 1)$; all parties store this locally in memory as the value Λ_{w_0}.

Open To open the circuit, the parties do the following:

1. For all $i \in [n]$, call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{Open}, 0, [\![\tilde{g}^j_{\alpha,\beta}]\!]_{2^k}, 1)$ for all $g \in G_{\text{AND}}$, for all $j \in [n]$, for all $(\alpha, \beta) \in \{0,1\}^2$. If the functionality returns $\perp$, the parties abort, and otherwise the parties (locally) output $((\tilde{g}^j_{0,0}, \tilde{g}^j_{0,1}, \tilde{g}^j_{1,0}, \tilde{g}^j_{1,1})^n_{j=1})_{g \in G}$ and the input mask identifiers $[\![r_1]\!]_p, \ldots, [\![r_t]\!]_p$.
2. Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{Check}, 1)$.

Evaluate Refer to $\Pi^{\mathsf{Eval}}_{\mathsf{ABB+BMR}}$ in Figure 13.

Fig. 11. Protocol $\Pi_{\mathsf{ABB+BMR}}$ (continued)

providing bits as input is equivalent to providing an entire $\mathbb{F}_p$ field element, strongly contrasting the case for $\mathbb{F}_{2^k}$; thus the cost for an input is 2.048kb. Hence, with the current state of protocols, inputs are cheap in a binary field whereas triples are cheap in a prime field.

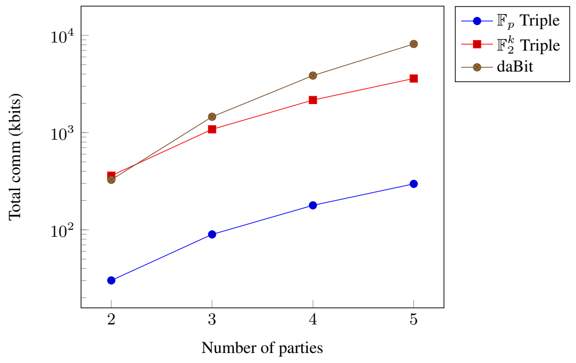

Fig. 14. Total communication costs for all parties per preprocessed element.

Bucketing parameters. Recall that our goal is to minimise the total amount of communication and time spent by parties generating each daBit. After examining the input and triple costs for LowGear and MASCOT (see Table 6 in Appendix D) we observed that the optimal communication for statistical security sec $= 64$ and a $p \approx 2^{128}$ is achieved with a generation of $l = 8192$ daBits per loop, a cut-and-choose parameter and $C = 5$ and a bucket size $B = 4$. Then we ran the daBit generation along with LowGear and MASCOT for multiple parties on the same computer configuration to get the total communication cost in order to see how communication scales in terms of number of parties. Results are given in Figure 14. Although MASCOT

Subprotocol $\Pi^{\mathsf{Garble}}_{\mathsf{ABB+BMR}}$

```
Garble  Traversing the circuit in topological order, for every gate $g \in G$ with (internal) input wires $u$ and $v$ and (internal) output wire $w$,
  – If $g$ is an XOR gate, i.e. $g \in G_{\text{XOR}}$,
    1. The parties set $[\![\lambda_w]\!]_{2^k} \leftarrow [\![\lambda_u]\!]_{2^k} \oplus [\![\lambda_v]\!]_{2^k}$.
    2. For each $i \in [n]$, $P_i$ computes $\mathsf{k}^i_{w,0} \leftarrow \mathsf{k}^i_{u,0} \oplus \mathsf{k}^i_{v,0}$ and $\mathsf{k}^i_{w,1} \leftarrow \mathsf{k}^i_{w,0} \oplus R^i$ and all parties set $[\![\mathsf{k}^i_{w,0}]\!]_{2^k} \leftarrow [\![\mathsf{k}^i_{u,0}]\!]_{2^k} \oplus [\![\mathsf{k}^i_{v,0}]\!]_{2^k}$ and $[\![\mathsf{k}^i_{w,1}]\!]_{2^k} \leftarrow [\![\mathsf{k}^i_{w,0}]\!]_{2^k} \oplus [\![R^i]\!]_{2^k}$.
  – If $g$ is an AND gate, i.e. $g \in G_{\text{AND}}$,
    1. The parties call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{RBit}, [\![\lambda_w]\!]_{2^k}, 1)$.
    2. For each $i \in [n]$,
      (a) Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{RElt}, [\![\mathsf{k}^i_{w,0}]\!]_{2^k}, 1)$.
      (b) Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{Open}, i, [\![\mathsf{k}^i_{w,0}]\!]_{2^k}, 1)$ to reveal $\mathsf{k}^i_{w,0}$ to $P_i$.
      (c) $P_i$ sets the one key as $\mathsf{k}^i_{w,1} \leftarrow \mathsf{k}^i_{w,0} \oplus R^i$ and all parties set $[\![\mathsf{k}^i_{w,1}]\!]_{2^k} \leftarrow [\![\mathsf{k}^i_{w,0}]\!]_{2^k} \oplus [\![R^i]\!]_{2^k}$.
      (d) For all four distinct values of $(\alpha, \beta) \in \{0,1\}^2$, and for every $j \in [n]$, $P_i$ calls $\mathcal{F}_{\mathsf{Prep}}$ with input $\left(\mathsf{Input}, i, [\![F^{g,i,j}_{\alpha,\beta}]\!]_{2^k}, F_{\mathsf{k}^i_{u,\alpha}, \mathsf{k}^i_{v,\beta}}(g \| j), 1\right)$ and the other parties with input $\left(\mathsf{Input}, i, [\![F^{g,i,j}_{\alpha,\beta}]\!]_{2^k}, \perp, 1\right)$.
    3. For all $j \in [n]$ and all $(\alpha, \beta) \in \{0,1\}^2$, the parties compute
```

$$[\![\tilde{g}^j_{\alpha,\beta}]\!]_{2^k} \leftarrow \left(\bigoplus_{i=1}^n \left[\!\left[F^{g,i,j}_{\alpha,\beta}\right]\!\right]_{2^k}\right) \oplus [\![\mathsf{k}^j_{w,0}]\!]_{2^k} \oplus [\![R^j]\!]_{2^k} \cdot \left(([\![\lambda_u]\!]_{2^k} \oplus \alpha) \cdot ([\![\lambda_v]\!]_{2^k} \oplus \beta) \oplus [\![\lambda_w]\!]_{2^k}\right)$$

Fig. 12. Subprotocol $\Pi^{\mathsf{Garble}}_{\mathsf{ABB+BMR}}$

triples are never used during the daBit production, we believe that comparing the cost of a daBit to the best triple generation in $\mathbb{F}_{2^k}$ helps to give a rough idea of how expensive a single daBit is.

	$\mathsf{sec} > 40$			$\mathsf{sec} > 64$			$\mathsf{sec} > 80$		
# daBits	128	1024	8192	128	1024	8192	128	1024	8192
Calls to $\mathcal{F}^{\{p,2^k\}}_{\mathsf{MPC}}$.**Input**	40	16	12	42	40	40	36	28	24
Calls to $\mathcal{F}^{p}_{\mathsf{MPC}}$.**Multiply**	7	7	5	13	9	7	17	13	11
Achieved sec	40	47	44	67	64	64	82	84	90

Table 1. Two parties pre-processing costs per daBit while varying the number of daBits per batch and statistical security. Parameters minimize for total communication given by LowGear and MASCOT.

To see how efficiency scales when the statistical security parameter sec is increased, we record the fewest numbers of calls to $\mathcal{F}_{\mathsf{MPC}}$, optimising for total (actual) communication cost in Table 1. Since the numbers are dependent on integers (number of parties, size of buckets, and cut and choose parameter), several of the numbers in the table give far better security than the minimum stated. Note that since we optimise for the total communication cost and not for the smallest **Cut and Choose** and **Bucketing** parameters that achieve each level of security, in the cost for $\mathsf{sec} = 64$ the number of calls to $\mathcal{F}_{\mathsf{MPC}}$.**Input** is larger than for $\mathsf{sec} = 80$. The bucket size, correlated with the number of calls to $\mathcal{F}_{\mathsf{MPC}}$.**Multiply**, is therefore is smaller than for $\mathsf{sec} = 80$.

sec	$\log p$	k	Comm. (kb)			Total (kb)	Time (ms)			Total(ms)
			$\mathcal{F}^{p}_{\mathsf{MPC}}$	$\mathcal{F}^{2^k}_{\mathsf{MPC}}$	daBitgen		$\mathcal{F}^{p}_{\mathsf{MPC}}$	$\mathcal{F}^{2^k}_{\mathsf{MPC}}$	daBitgen	
40	128	128	76.60	2.30	6.94	85.84	0.159	$<$ 10ns	0.004	0.163
64	128	128	146.47	7.68	9.39	163.54	0.303	$<$ 10ns	0.010	0.313
80	128	128	192.95	4.60	7.32	204.88	0.485	$<$ 10ns	0.008	0.493

Table 2. 1Gb/s LAN experiments for two-party daBit generation per party. For all cases, the daBit batch has length 8192.

4.1 Share conversion

To reduce the amount of garbling when converting an additive share to a GC one, if we assume the $\mathbb{F}_p$ input to the garbled circuit is bounded by $p/2^{\mathsf{sec}}$, then a uniform r in $\mathbb{F}_p$ is 2^{sec} times larger than a so $a - r$ is statistically-indistinguishable from a uniform element of $\mathbb{F}_p$; consequently, one need only garble $a + r$ and not $a + r \mod p$, which makes the circuit marginally smaller – 379 AND gates for a 128 bit prime rather than $\approx$ 1000 AND gates for an addition mod p circuit.

In Table 3 we split the conversion into two phases: the total cost of generating 127 daBits for doing a full conversion (including the preprocessing triples from LowGear) and the online phase of SPDZ-BMR.

Comparison to semi-honest conversion. When benchmarked with 40 bit statistical security, the online phase to convert 1000 field elements of size 32 bits takes 193ms. Our solution benefits from merging multiple conversions at once due to the SIMD nature of operations and that we can perform a single MAC-Check to compute the signal bits for the GC. Note that our conversion from an arithmetic SPDZ share to a SPDZ-BMR GC share takes about 14 times more than the semi-honest arithmetic to an Yao GC conversion in ABY or Chameleon on an identical computer configuration [RWT+18, DSZ15].

4.2 Multiple class Support Vector Machine

We have benchmarked the online phase of a multi-class Linear SVM (Support Vector Machine) with 100 classes and 128 features over a simulated WAN network with a round-trip ping time of 100ms and 50Mb/s bandwidth) with two parties. This is the same SVM structure used by Makri et al. [MRSV17] to classify the Caltech-101 dataset which contains 102 different categories of images such as aeroplanes, dolphins, helicopters and others [FFFP04].

An SVM is just a simple computation of $\mathsf{argmax}([\![A]\!][\![\mathbf{x}]\!] + [\![b]\!])$ (i.e. the index of the first component of the vector $[\![A]\!][\![\mathbf{x}]\!] + [\![b]\!]$ attaining $||[\![A]\!][\![\mathbf{x}]\!] + [\![b]\!]||_\infty$) where the where the feature vector $[\![\mathbf{x}]\!]$ has length 128,

Conversion	daBit (total)		SPDZ-BMR	
	Comm. (kbits)	Time (ms)	ANDs	Online (ms)
SPDZ $\mapsto$ GC	20769	39.751	379	0.106
GC $\mapsto$ SPDZ	10303	19.719	0	0.005

Table 3. Two parties 1Gb/s LAN experiments converting a 63 bit field element with 64 statistical security. BMR online phase times are amortized over 1000 executions in parallel (single-threaded).

the matrix $[\![A]\!] \in \mathbb{F}_p^{102\times 128}$ and vector $[\![b]\!] \in \mathbb{F}_p^{102}$. The particular SVM used by Makri et al. has bounded inputs x where $\log|x| \leq 25$, a field size $\log p = 128$ and statistical security $\mathsf{sec} = 64$. We chose this particular circuit because it is easy to see which part has to be done with arithmetic sharing ($[\![A]\!] \cdot [\![\mathbf{x}]\!] + [\![b]\!]$) and which part within GC (argmax).

We have implemented a special instruction in MP-SPDZ which loads a secret integer modulo p (a SPDZ share) into the SPDZ-BMR machine. To merge all modulo p instructions of SPDZ shares into SPDZ-BMR to form an universal Virtual Machine requires some extra engineering effort: this is why we chose to benchmark in Table 4 the different stages of the online phase: doing $[\![y]\!]_p \leftarrow [\![A]\!][\![\mathbf{x}]\!] + [\![b]\!]$ with SPDZ, then the conversion instruction on $[\![y]\!]_{2^k} \leftarrow \mathsf{switch}([\![y]\!]_p)$, ending with the evaluation stage of SPDZ-BMR on $\mathsf{argmax}([\![y]\!]_{2^k})$. We name this construction Marbled-SPDZ.

The results are given in Table 4, where the sum of the last three columns gives the total time for Marbled-SPDZ. The online phase using Marbled-SPDZ is more than 10 times faster than SPDZ-BMR and about 10 times faster than SPDZ. The preprocessing effort for the garbling (AND gates) is reduced by a factor of almost 400 times. Unfortunately, the triple generation effort for SPDZ increases by a factor of 2.6.

Protocol	Sub-Protocol	Online cost			Preprocessing cost		
		Comm. rounds	Time (ms)	Total (ms)	$\mathbb{F}_p$ triples	$\mathbb{F}_p$ bits	AND gates
SPDZ		54	2661	2661	19015	9797	-
SPDZ-BMR		0	2786	2786	-	-	14088217
Marbled-SPDZ	SPDZ	1	133	271.73	13056	0	-
	daBit convert	2	137		63546	0	27030
	SPDZ-BMR	0	1.73		-	-	8383

Table 4. Two-party linear SVM: single-threaded (non-amortized) online phase costs and preprocessing costs with $\mathsf{sec} = 64$.

Instantiating the preprocessing. As described in Section 2, since our method works independently of the underlying GC scheme we can instantiate it with more efficient garbling protocols such as Wang et al. [WRK17] or Hazay et al. [HSS17]. To estimate the total communication effort done by Marbled-SPDZ for the GC part in the preprocessing we used Wang et al. measurements for 40 bit statistical security. In order to give a fair comparison we also take daBits which are generated with $\mathsf{sec} = 40$ (see Table 2). We also set the plain online phase of SPDZ to perform comparisons with 40 bit statistical security. Lowering this parameter for SPDZ online comparisons also reduces the amount of random shared bits needed, hence a smaller preprocessing cost.

The online phase masking is done with $25 + 40$ daBits instead of $25 + 64$ daBits as Table 4. This takes only 193 AND gates to remove the randomness inside the GC, as opposed to 265 ANDs as in Table 4. We can see that the total communication done by Marbled-SPDZ when instantiated with WRK garbling is two times larger than performing plain SPDZ. This seems to be a trade-off that one has to pay in order to have a constant round protocol in the online phase.

Circuit type	Sub-Protocol	Preprocessing protocol (comm.)			Total
		LowGear	WRK (indep.)	WRK (dep.)	
SPDZ		49.4 MB	-	-	49.4 MB
GC		-	4917 MB	1768 MB	6685 MB
Marbled	SPDZ	24.48 MB	-	-	108.87 MB
	daBit convert	71.13 MB	6.83 MB	2.45 MB	
	GC	-	2.92 MB	1.05 MB	

Table 5. Two-party linear SVM communication cost for preprocessing in MBytes and statistical security $\mathsf{sec} = 40$.

Acknowledgements

We would like to thank COED at COSIC at KU Leuven, and Emmanuela Orsini, Nigel Smart and Younes Talibi in particular, as well as Marcel Keller and Peter Scholl, for providing their enlightening insight and suggestions. Special thanks goes to Marcel Keller who helped with the understanding of SPDZ-BMR code in the MP-SPDZ framework.

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 by the FWO under an Odysseus project GOH9718N.

Subprotocol $\Pi^{\mathsf{Eval}}_{\mathsf{ABB+BMR}}$

Evaluate The parties, holding the output $((\widetilde{g}^i_{0,0},\ \widetilde{g}^i_{0,1},\ \widetilde{g}^i_{1,0},\ \widetilde{g}^i_{1,1})^n_{i=1})_{g\in G}$ of $\mathcal{F}^{\mathsf{Garble+}}_{\mathsf{BMR}}$, evaluate in the following way, traversing the circuit in topological order:

1. For each input $\{[\![a_i]\!]_p\}_{i\in[t]}$, the parties do the following:
   (a) Retrieve from memory the secret mask $[\![r_i]\!]_p$ produced in **Input layer**.
   (b) Compute the secret $[\![x_i]\!]_p \leftarrow [\![a_i]\!]_p - [\![r_i]\!]_p$.
   (c) Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{Open}, [\![x_i]\!]_p, 0)$.
   (d) Denote the corresponding input wires by $\{w_{i,j}\}_{j=0}^{\lfloor\log p\rfloor-1}$. Bit-decompose the public value x_i and let the bits be $\{x_{i,j}\}_{j=0}^{\lfloor\log p\rfloor-1}$.
   (e) For each $j = 0,\ldots,\lfloor\log p\rfloor - 1$, retrieve from memory the wire mask $\lambda_{w_{i,j}}$ from **Input layer** for $x_{i,j}$ and set $\Lambda_{w_{i,j}} \leftarrow x_{i,j} \oplus \lambda_{w_{i,j}}$.
   (f) For each $l \in [n]$, P_l sends $\{\mathsf{k}^l_{w_{i,j},\Lambda_{w_{i,j}}}\}_{j=0}^{\lfloor\log p\rfloor-1}$ to all other parties.
2. For every $g \in G$,
   (a) If g is an XOR gate,
      i. Party P_i computes $\Lambda_w \leftarrow \Lambda_u \oplus \Lambda_v$.
      ii. Party P_i computes all n output keys indexed by $j \in [n]$, as $\mathsf{k}^j_{w,\Lambda_w} \leftarrow \mathsf{k}^j_{u,\Lambda_u} \oplus \mathsf{k}^j_{v,\Lambda_v}$.
   (b) If g is an AND gate,
      i. Each party computes the n keys indexed by $j \in [n]$ as

$$\mathsf{k}^j_{w,\Lambda_w} \leftarrow \widetilde{g}^j_{\Lambda_u,\Lambda_v} \oplus \left(\bigoplus_{i=1}^{n} F_{\mathsf{k}^i_{u,\Lambda_u},\mathsf{k}^i_{v,\Lambda_v}}(g\|j)\right)$$

      and compares its keys $\mathsf{k}^i_{w,0}$ and $\mathsf{k}^i_{w,1}$ to the i^{th} key obtained to determine the global signal bit Λ_w.
3. For every external output wire w,
   (a) Retrieve from memory the corresponding public signal bit Λ_{w_0} produced in **Output layer**.
   (b) Locally compute $\Lambda_{w'} \leftarrow \Lambda_{w_0} \oplus \Lambda_w$.
   (c) Locally compute the secret output as

$$[\![b_w]\!]_p \leftarrow \Lambda_{w'} + [\![\lambda_{w'}]\!]_p - 2\cdot\Lambda_{w'}\cdot[\![\lambda_{w'}]\!]_p.$$

4. Send the message $(\mathsf{Check}, 1)$ to $\mathcal{F}_{\mathsf{Prep}}$.

Fig. 13. Subprotocol $\Pi^{\mathsf{Eval}}_{\mathsf{ABB+BMR}}$

References

[ABF$^+$18] Toshinori Araki, Assi Barak, Jun Furukawa, Marcel Keller, Yehuda Lindell, Kazuma Ohara, and Hikaru Tsuchida. Generalizing the spdz compiler for other protocols. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 880–895. ACM, 2018.

[AKO$^+$18] A Aly, M Keller, E Orsini, D Rotaru, P Scholl, N Smart, and T Wood. Scale-mamba v1.3 : Documentation, 2018. https://homes.esat.kuleuven.be/~nsmart/SCALE/.

[Ana19] N1 Analytics. MP-SPDZ, 2019. https://github.com/n1analytics/MP-SPDZ.

[BE17] Aner Ben-Efraim. On multiparty garbling of arithmetic circuits. Cryptology ePrint Archive, Report 2017/1186, 2017. https://eprint.iacr.org/2017/1186.

[Bea92] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 420–432. Springer, Heidelberg, August 1992.

[Ben17] Aner Ben-Efraim. On multiparty garbling of arithmetic circuits. Cryptology ePrint Archive, Report 2017/1186, 2017. https://eprint.iacr.org/2017/1186.

[BGV11] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. Cryptology ePrint Archive, Report 2011/277, 2011. http://eprint.iacr.org/2011/277.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd ACM STOC*, pages 503–513. ACM Press, May 1990.

[BMR16] Marshall Ball, Tal Malkin, and Mike Rosulek. Garbling gadgets for boolean and arithmetic circuits. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 565–577. ACM Press, October 2016.

[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.

[CDE$^+$18] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 769–798. Springer, Heidelberg, August 2018.

[CKKZ12] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. On the security of the "free-XOR" technique. In Ronald Cramer, editor, *TCC 2012*, volume 7194 of *LNCS*, pages 39–53. Springer, Heidelberg, March 2012.

[DFK$^+$06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography Conference*, pages 285–304. Springer, 2006.

[DKL$^+$13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P Smart. Practical covertly secure mpc for dishonest majority–or: breaking the spdz limits. In *European Symposium on Research in Computer Security*, pages 1–18. Springer, 2013.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology–CRYPTO 2012*, pages 643–662. Springer, 2012.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS 2015*. The Internet Society, February 2015.

[DZ13] Ivan Damgård and Sarah Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In *Theory of Cryptography*, pages 621–641. Springer, 2013.

[FFFP04] Li Fei-Fei, R Fergus, and P Perona. Learning Generative Visual Models from Few Training Examples: An Incremental Bayesian Approach Tested on 101 Object Categories. In *CVPR*, pages 178–178. IEEE, 2004.

[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, *ASIACRYPT 2015, Part I*, volume 9452 of *LNCS*, pages 711–735. Springer, Heidelberg, November / December 2015.

[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, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 598–628. Springer, Heidelberg, December 2017.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 830–842. ACM, 2016.

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 158–189. Springer, Heidelberg, April / May 2018.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In *International Colloquium on Automata, Languages, and Programming*, pages 486–498. Springer, 2008.

[KY18] Marcel Keller and Avishay Yanai. Efficient maliciously secure multiparty computation for RAM. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 91–124. Springer, Heidelberg, April / May 2018.

[LPSY15] Yehuda Lindell, Benny Pinkas, Nigel P Smart, and Avishay Yanai. Efficient constant round multi-party computation combining BMR and SPDZ. In *Annual Cryptology Conference*, pages 319–338. Springer, 2015.

[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.

[MRSV17] Eleftheria Makri, Dragos Rotaru, Nigel P. Smart, and Frederik Vercauteren. Epic: Efficient private image classification (or: Learning from the masters). Cryptology ePrint Archive, Report 2017/1190, 2017. https://eprint.iacr.org/2017/1190.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *Advances in Cryptology–CRYPTO 2012*, pages 681–700. Springer, 2012.

[RWT$^+$18] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim, editors, *ASIACCS 18*, pages 707–721. ACM Press, April 2018.

[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 17*, pages 39–56. ACM Press, October / November 2017.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

A Various Functionalities and Protocols

In Figures 15, 16, 17 and 18 we give some of the standard functionalities and protocols securely realising them in the UC framework. We omit the proofs as they are standard.

Protocol Π_{Rand}

This protocol is in the $\mathcal{F}_{\mathsf{Commit}}$-hybrid model. Let $\mathsf{RShuffle}(\mathsf{seed}, s)$ denote any deterministic algorithm that takes a random seed seed and a vector s and outputs a permutation of components of s. Recall κ is the computational security parameter.

Initialise Parties agree on a session identifier sid and call $\mathcal{F}_{\mathsf{Commit}}$ with input $(\mathsf{Initialise}, \{0,1\}^\kappa, \mathsf{sid})$.

Seed This is a subroutine. Parties do the following:

1. For each $i \in [n]$, P_i samples $\mathsf{seed}_i \xleftarrow{\$} \{0,1\}^\kappa$.
2. For each $i \in [n]$, P_i sends the message $(\mathsf{Commit}, i, \mathsf{seed}_i, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{Commit}}$ and all other parties send $(\mathsf{Commit}, i, \perp, \mathsf{sid})$; all parties receive $\mathsf{id}_{\mathsf{seed}_i}$ in response.
3. All parties call $(\mathsf{Open}, i, \mathsf{id}_{\mathsf{seed}_i}, \mathsf{sid})$ to obtain $\{\mathsf{seed}_i\}_{i\in[n]}$.
4. P_i sets $\mathsf{seed} \leftarrow \bigoplus_{i=1}^n \mathsf{seed}_i$.

Random subset To compute a random subset of size t of a set X, parties run **Seed** to obtain a seed seed for a PRG and then do the following:

1. Let $X = \{x_i\}_{i=1}^{|X|}$. Parties set the vector $s = (s_1, \ldots, s_{|X|}) \leftarrow (1, \ldots, 1, 0, \ldots, 0) \in \{0,1\}^{|X|}$, where the first t bits are set to 1 and the remaining bits set to 0.
2. Each P_i locally computes $s' = (s'_1, \ldots, s'_{|X|}) \leftarrow \mathsf{RShuffle}(\mathsf{seed}, s)$ and outputs the set $S \leftarrow \{x_i : s'_i = 1\}$.

Random buckets To put a set of items indexed by a set X into buckets of size t where t divides $|X|$, parties run **Seed** to obtain a seed seed for a PRG and then do the following:

1. Let $X = \{x_i\}_{i=1}^{|X|}$. Each P_i locally computes $s' \leftarrow \mathsf{RShuffle}(\mathsf{seed}, s)$ where $s \leftarrow (i)_{i=1}^{|S|}$.
2. For each $i = 1$ to $|S|/t$, let $S_i \leftarrow \{x_{s'_j} : (i-1)\cdot t < j \leq i \cdot t\}$.

Fig. 15. Protocol Π_{Rand}

B SPDZ-BMR PRF Assumption

The non-existence of *circular* 2*-correlation robust* PRFs required for using the multiparty FreeXOR technique would force garbling protocols to garble XOR gates in the same way as AND gates, providing PRFs under the (supposed) weaker assumption of *pseudorandom function under multiple keys* exist. These are defined as follows:

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

The functionality keeps track of the current session using a session identifier sid. If a party provides an input with sid different from what was sent in **Initialise**, the functionality outputs Reject to all parties and awaits another message. Recall that κ is the computational security parameter.

Initialise On input $(\mathsf{Initialise}, X, \mathsf{sid})$ from all parties where X is a set, initialise a dictionary of values Val with identifiers $\mathsf{Val.Keys}$.

Commit On input $(\mathsf{Commit}, i, x, \mathsf{sid})$ from P_i, or the adversary if P_i is corrupt, and $(\mathsf{Commit}, i, \perp, \mathsf{sid})$ from all other parties, where $x \in X$, choose new identifier $\mathsf{id}_x \overset{\$}{\leftarrow} \{0,1\}^\kappa$, add id_x to $\mathsf{Val.Keys}$, set $\mathsf{Val}[\mathsf{id}_x] \leftarrow x$, and send id_x to all parties.

Open On input $(\mathsf{Open}, i, \mathsf{id}_x, \mathsf{sid})$ from all parties where $\mathsf{id}_x \in \mathsf{Val.Keys}$, if P_i is corrupt then await a message OK or Reject from the adversary. If the message is OK or P_i is honest then send $\mathsf{Val}[\mathsf{id}_x]$ to all parties and otherwise halt.

Fig. 16. Functionality $\mathcal{F}_{\mathsf{Commit}}$

Definition 2. *Let $F : \{0,1\}^\kappa \times \{0,1\}^\kappa \to \{0,1\}^\kappa$ be an efficient, length-preserving, keyed function. We say that F is a* pseudorandom function under multiple keys *if for all polynomial time distinguishers $\mathcal{D}$ there exists a negligible function ν such that:*

$$\left| \Pr[\mathcal{D}^{F_{\bar{\mathsf{k}}}(\cdot)}(1^{\mathsf{sec}}) = 1] - \Pr[\mathcal{D}^{\bar{f}(\cdot)}(1^{\mathsf{sec}}) = 1] \right| \leq \nu(\kappa).$$

where $F_{\bar{\mathsf{k}}}$ denotes the tuple $(F_{\mathsf{k}_1}, \ldots, F_{\mathsf{k}_n})$ of the pseudorandom function F keyed using $\mathsf{k}_1, \ldots, \mathsf{k}_n$ and $\bar{f}$ denotes the tuple $(f_1, \ldots, f_n)$ of random functions $\{f_i : \{0,1\}^\kappa \to \{0,1\}^\kappa\}_{i=1}^n$.

Lindell et al. proved that the SPDZ-BMR technique is secure under this assumption on the PRF.

C Proof of Theorem 2

Proof. We define a simulator in Figure 19. Note that the only secrets to which the simulator is not privy are the secret inputs of honest parties. Everything else in the protocol involves calls to $\mathcal{F}_{\mathsf{Prep}}$ which is locally emulated by the simulator.

We define the hybrids in the same way as in the proof of Theorem 1. Suppose the adversary corrupts $t < n$ parties in total, indexed by a set A. We define a sequence of hybrid worlds $(\textbf{Hybrid } h)_{h=0}^{n-t}$ and show that each is indistinguishable from the previous. **Hybrid** h is defined as:

Hybrid h The simulator has the actual input of $n - t - h$ honest parties and must simulate the remaining h honest parties towards the adversary.

The simulator is described in Figure 19, parameterised by h.

Claim. The $\mathcal{F}_{\mathsf{Prep}}$-hybrid world is indistinguishable from **Hybrid** 0.

Protocol Π_{Commit}

This protocol is in the $\mathcal{F}_{\mathsf{RO}}$-hybrid model (ROM). Recall that κ is the computational security parameter. We write $a\|b$ to mean a concatenated with b.

Initialise The parties agree on a session identifier sid and call $\mathcal{F}_{\mathsf{RO}}$ with input $(\mathsf{Initialise}, \{0,1\}^\kappa, \mathsf{sid})$.
Commit For P_i to commit to x, the parties do the following:

1. P_i samples $r \xleftarrow{\$} \{0,1\}^\kappa$
2. P_i calls $\mathcal{F}_{\mathsf{RO}}$ with input $(\mathsf{Query}, x\|r, \mathsf{sid})$, receives h_x in response and broadcasts it.
3. All other parties store h_x locally.

Open For P_i to open the commitment to x, the parties do the following:

1. P_i broadcasts x and r.
2. P_j calls $\mathcal{F}_{\mathsf{RO}}$ with input $(\mathsf{Query}, x\|r, \mathsf{sid})$ and checks that the response is equal to h_x received during commitment.

Fig. 17. Protocol Π_{Commit}

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

The functionality keeps track of the current session using a session identifier sid. If a party provides an input with sid different from what was sent in **Initialise**, the functionality outputs Reject to all parties and awaits another message.

Initialise On input $(\mathsf{Initialise}, X, \mathsf{sid})$ from all parties, initialise a dictionary of values Val.
Random Element On input $(\mathsf{Query}, q, \mathsf{sid})$ from party P_i where $q \in \{0,1\}^*$, if $\mathsf{Val}[q]$ has not yet been defined, uniformly sample $\mathsf{Val}[q] \xleftarrow{\$} X$ and send $\mathsf{Val}[q]$ to P_i.

Fig. 18. Functionality $\mathcal{F}_{\mathsf{RO}}$

Proof. For the emulation of $\mathcal{F}_{\mathsf{Prep}}$ with $\mathsf{sid} = 0$, the simulation is perfect.

For the emulation of $\mathcal{F}_{\mathsf{Prep}}$ with $\mathsf{sid} = 1$, there are no private inputs of honest parties to the garbling, so it only remains to show that the transcript during evaluation reveals nothing about the (honest) parties' inputs.

In this hybrid, the simulator has access to all honest parties' inputs, so the simulator follows the protocol exactly in **Evaluate**, so the worlds are indistinguishable. ■

Claim. **Hybrid** h is indistinguishable from **Hybrid** $h+1$ for $h = 0, \dots, n-t-1$.

Proof. For the emulation of $\mathcal{F}_{\mathsf{Prep}}$ with $\mathsf{sid} = 0$ and for the garbling the simulation is still perfect since honest parties' inputs are not required.

Indeed, the only call to **Input** is when when the parties call $\mathcal{F}_{\mathsf{Prep}}$ during **Garble** for the PRF evaluations. The simulator can perform the PRF evaluations locally since the keys and global differences for honest parties are obtained from the emulation of $\mathcal{F}_{\mathsf{Prep}}$.

Simulator $\mathcal{S}_{\mathsf{ABB+BMR}}$

Let H_R denote the indexing set of the h honest parties whose inputs are known to $\mathcal{S}^h_{\mathsf{ABB+BMR}}$. Initialise and run an internal copy of $\mathcal{F}_{\mathsf{Prep}}$ with the adversary, answering every query by executing the code of $\mathcal{F}_{\mathsf{Prep}}$. For simplicity of relaying messages between $\mathcal{A}$ and $\mathcal{F}_{\mathsf{CABB}}$, we assume $\mathsf{sid} = 0$ for $\mathcal{F}_{\mathsf{CABB}}$ as well as the $\mathbb{F}_p$ instance of $\mathcal{F}_{\mathsf{Prep}}$.

Initialise Initialise a local copy of $\mathcal{F}_{\mathsf{Prep}}$ and await the inputs $(\mathsf{Initialise}, \mathbb{F}_p, 0)$ and $\mathsf{Initialise}, \mathbb{F}_{2^k}, 1)$ from $\mathcal{A}$.

ABB For calls to $\mathcal{F}_{\mathsf{Prep}}$ with $\mathsf{sid} = 0$:

- **Input** On input $(\mathsf{Input}, i, \mathsf{id}, 0)$, forward the message to $\mathcal{F}_{\mathsf{CABB}}$.
- **Add** On input $(\mathsf{Add}, \mathsf{id}_x, \mathsf{id}_y, \mathsf{id}, 0)$, forward the message to $\mathcal{F}_{\mathsf{CABB}}$.
- **Multiply** On input $(\mathsf{Multiply}, \mathsf{id}_x, \mathsf{id}_y, \mathsf{id}, 0)$, forward the message to $\mathcal{F}_{\mathsf{CABB}}$.
- **Output**
  1. Await a message $(\mathsf{Check}, 0)$ from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{Prep}}$ and then await a message $(\mathsf{OK}, 0)$ or $(\mathsf{Abort}, 0)$ from $\mathcal{A}$, and if it is $(\mathsf{Abort}, 0)$ then ignore all further calls to $\mathcal{F}_{\mathsf{Prep}}$ with $\mathsf{sid} = 0$ and otherwise continue.
  2. On input $(\mathsf{Open}, 0, \mathsf{id}, 0)$ to $\mathcal{F}_{\mathsf{Prep}}$, send the message $(\mathsf{Output}, \mathsf{id}, 0)$ to $\mathcal{F}_{\mathsf{CABB}}$ and relay the response x to $\mathcal{A}$.
  3. Await a reply $x + \varepsilon$ from $\mathcal{A}$ and the call by $\mathcal{A}$ to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{Check}, 0)$.
  4. Await another message $(\mathsf{OK}, 0)$ or $(\mathsf{Abort}, 0)$ from $\mathcal{A}$. If $\varepsilon = 0$ and the message was $(\mathsf{OK}, 0)$ then send OK to $\mathcal{F}_{\mathsf{CABB}}$, and otherwise send Abort to $\mathcal{F}_{\mathsf{CABB}}$ and $(\mathsf{Abort}, 0)$ to $\mathcal{A}$ and ignore all further calls to $\mathcal{F}_{\mathsf{Prep}}$ with $\mathsf{sid} = 0$ and otherwise continue.

Initialise garbling Run **Initialise** from $\Pi_{\mathsf{ABB+BMR}}$ with $\mathcal{A}$.

Input layer Run **Input layer** from $\Pi_{\mathsf{ABB+BMR}}$ with $\mathcal{A}$.

Garble Run $\Pi^{\mathsf{Garble}}_{\mathsf{ABB+BMR}}$ with $\mathcal{A}$.

Output layer Run **Output layer** from $\Pi_{\mathsf{ABB+BMR}}$ with $\mathcal{A}$.

Open Run **Open** from $\Pi_{\mathsf{ABB+BMR}}$ with $\mathcal{A}$.

Evaluate Suppose a circuit input x is from an honest party's input.

1. Send the message $(\mathsf{EvaluateCircuit}, C, \mathsf{id}_1, \dots, \mathsf{id}_t, \mathsf{id}, 0)$ to $\mathcal{F}_{\mathsf{CABB}}$.
2. Await the call $(\mathsf{Open}, 0, [\![a - r]\!])$ from $\mathcal{A}$. Then:
   - If a is an input dependent on honest parties' inputs known to $\mathcal{S}^h_{\mathsf{ABB+BMR}}$, retrieve the bits of the mask r from memory and compute and send $x \leftarrow a - r$ to the adversary.
   - If a is dependent on one or more honest parties' inputs then sample $x \leftarrow r' \xleftarrow{\$} \mathbb{F}_p$ and send it to $\mathcal{A}$.
3. Await a response $x+\varepsilon$ and if $\varepsilon \neq 0$ then send $(\mathsf{Abort}, 0)$ to $\mathcal{A}$ and Abort to $\mathcal{F}_{\mathsf{CABB}}$ and terminate; otherwise, continue.
4. Await the calls to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathsf{Open}, 0, [\![\mathsf{k}_{u_j, x_j}]\!]_{2^k}, 1)_{j=0}^{\lceil \log p \rceil - 1}$ from $\mathcal{A}$, where u_j is the wire for the jth input bit x_j of x, and respond honestly.
5. The simulator computes what honest parties would compute in the circuit evaluation. If an honest party would have aborted then the simulator sends Abort to $\mathcal{F}_{\mathsf{CABB}}$, and otherwise sends OK and continues.

Fig. 19. Simulator $\mathcal{S}^h_{\mathsf{ABB+BMR}}$

The only part of the circuit evaluation that may depend on honest parties' inputs is in **Evaluate**. Since the secret masks $[\![r]\!]_p$ are constructed from uniformly-sampled bits, by Lemma 1 the distribution of the uniformly sample $r' \overset{\$}{\leftarrow} \mathbb{F}_p$ is statistically close to the distribution of $a - r \mod p$ where a is the input of an honest party and $r \overset{\$}{\leftarrow} \{0,1\}^{\lfloor \log p \rfloor}$.

Now since the masking bits are sampled uniformly during the garbling and are unknown to the adversary (or environment), and the circuit can only be evaluated once, the intermediate wire values and the final bit (masked) outputs reveal nothing about the initial inputs to the circuit. Indeed, after evaluating the circuit the parties just have an identifier corresponding to the circuit output, which reveals no information on the underlying value by definition of the functionality. Thus the environment cannot use the evaluation to distinguish between the circuit evaluated on the actual value $a - r$ and the circuit evaluated on the sampled value random r', even though the parties will obtain an "incorrect" output identifier with high probability. The simulator easily deals with this by obtaining the output from $\mathcal{F}_{\mathsf{CABB}}$ and sending this to $\mathcal{A}$, ensuring that if the evaluation of the garbled circuit did caused an honest party to abort then the interaction with $\mathcal{A}$ and $\mathcal{F}_{\mathsf{CABB}}$ also abort. ■

Since $\mathcal{F}_{\mathsf{Prep}}$ is secure for $t = n - 1$, the result follows. □

D MASCOT and LowGear

Here we provide more data on the communcation complexity of these two protocols benchmarked for fields $\mathbb{F}_p$ and $\mathbb{F}_{2^k}$ where $k = 128$, $\log p > 128$, and statistical security sec $= 64$.

# Parties	MASCOT $\mathbb{F}_{2^k}$		LowGear $\mathbb{F}_p$	
	Input (bit)	Triple	Input	Triple
2	0.384	360.44	2.048	30.146
3	1.024	1081.32	5.888	89.67
4	1.92	2162.64	11.520	178.572
5	3.072	3604.4	18.94	296.85

Table 6. Communication costs (kbits) for fields with different characteristic.