

Compilation of Function Representations for Secure Computing Paradigms

Karim Baghery¹ , Cyprien Delpech de Saint Guilhem¹ , Emmanuela Orsini¹ , Nigel P. Smart^{1,2} , and Titouan Tanguy¹ 

¹ imec-COSIC, KU Leuven, Leuven, Belgium.

² University of Bristol, Bristol, UK.

karim.baghery@kuleuven.be, cyprien.delpechdesaintguilhem@kuleuven.be,
emmanuela.orsini@kuleuven.be, nigel.smart@kuleuven.be, titouan.tanguy@kuleuven.be

Abstract. This paper introduces M-Circuits, a program representation which generalizes arithmetic and binary circuits. This new representation is motivated by the way modern multi-party computation (MPC) systems based on linear secret sharing schemes actually operate. We then show how this representation also allows one to construct zero knowledge proof (ZKP) systems based on the MPC-in-the-head paradigm. The use of the M-Circuit program abstraction then allows for a number of program-specific optimizations to be applied generically. It also allows to separate complexity and security optimizations for program compilation from those for application protocols (MPC or ZKP).

1 Introduction

Secure computation methodologies are becoming more mainstream with multi-party computation (MPC), fully homomorphic encryption (FHE) and zero-know-ledge proofs of knowledge (ZKPoKs) all finding applications at an increasing rate. At their heart all three technologies work with a public function; either to be compute it securely (in the case of MPC or FHE), or to prove the correctness of public outputs under secret inputs (in the case of ZKPoKs). The representation of this function is key to many of the practical realizations. For example: in theoretical MPC papers functions are often represented by arithmetic circuits, in FHE they are often binary circuits, and in ZKPoK papers R1CS representations are often used.

In this work we concentrate on two secure computation technologies: MPC protocols based on linear secret sharing schemes (LSSS) and ZKPoKs based on MPC-in-the-Head (MPCitH). It is common in theoretical treatments of these protocols to assume the input function representation is given as an arithmetic or binary circuit. However, in practice, this is not how functions are represented as input to such protocols. It has been known since Beaver’s work [Bea92] that sometimes a more interesting representation is as a set of linear operations, combined with a correlated randomness source. Previous work showed that more efficient representations for LSSS-based MPC can be obtained using different sources of correlated randomness, as well as a combination of different finite fields [DFK⁺06, CS10, Cd10]. This latter idea has been expanded in recent years with the advent of so-called daBit-based protocols for switching between LSSS-based MPC and garbled circuit-based MPC [RW19]. Thus the standard theoretical assumption of representing the function as a simple arithmetic circuit is at least ten years out of date given the state-of-the-art of LSSS-based MPC protocols.

A similar situation holds for MPCitH protocols. These are a class of zero-knowledge protocols introduced by Ishai et al in [IKOS07], and recently extended in a number of works eg.

[AHIV17, BN20, BFH⁺20, CDG⁺17, KKW18, GMO16] In such protocols, a key aspect is to represent the function as a sequence of linear operations, combined with access to sources of correlated randomness, as in [KKW18, BN20]. In MPC protocols, the creation of the correlated randomness sources often involves expensive pre-processing, but for MPCitH this can be done essentially for free. Therefore, larger performance improvements for MPCitH could result from expanding the use of correlated randomness sources.

Indeed in both LSSS-based MPC and MPCitH there is no single ‘correct’ representation of a function, with different representations presenting different performance trade-offs in the final protocol. But it is also case that different representations can also present different security trade-offs as well. Indeed the compilation of the abstract function into a concrete representation can introduce security issues.

Our Contributions and Paper Overview. The first contribution of this paper (in Section 3) is a generalized definition of the program input to such LSSS-based MPC and MPCitH protocols, which we call an M-Circuit. It can be considered as a generalization of arithmetic circuits, but tuned for MPC and MPCitH protocols (hence the name). An M-Circuit can make use of linear operations on sensitive variables, correlated randomness sources, as well as objects we call ‘gadgets’. A gadget is a function call from the M-Circuit representation of the program which is not necessarily implemented itself as an M-Circuit. Such gadgets allow for specific functions to be implemented in ways which avoid inefficiencies from implementing them using only an M-Circuit definition. Each M-Circuit belongs to a set of classes of M-Circuits, determined by what correlated randomness sources we allow, what finite fields are utilized, and what magic ‘gadgets’ are used. We can then determine which classes of protocols are best suited for different protocols.

We furthermore examine what it means for a compilation of a function into an M-Circuit to be ‘secure’. And we relate this security definition to the security of the resulting MPC/MPCitH protocol when the given M-Circuit representation is used. Some compilation strategies are clearly insecure, some give perfect security and some give statistical security; leading to the same corresponding security of the final protocol in which they are used. Whilst well understood in the practical community, we can find no treatment of the security aspect of program compilation being discussed in the MPC literature before.

After presenting our representation we show how this maps, in Section 4, onto common MPC frameworks (such as MP-SPDZ [Kel20] and SCALE-MAMBA [ACK⁺20]), and how the M-Circuit representation is already the underlying one used in practice. This application is now standard and we only sketch it in this work.

In our second contribution in Section 5, we show how M-Circuits can be used in MPCitH protocols. We recast the protocols of [KKW18, BN20] to use our general M-Circuit definition (initially excluding the use of the gadgets). We define the components needed to allow M-Circuits to be used in general MPCitH protocols. We then go on to present a number of optimization strategies which our M-Circuit representation allows one to express easily for MPCitH protocols. The first, in Section 6, examines how introducing new correlated randomness sources can produce more efficient proofs. We recall that for MPCitH adding new correlated randomness sources comes at little extra cost and thus this is a resource we can utilize to improve efficiency quite aggressively. The second, in Section 7, shows how, for some randomness sources, one can replace cut-and-choose checking with a form of sacrificing (as used in MPC protocols such as SPDZ). This acts as a warm up for our method which introduces the ability to introduce complex gadgets into our MPCitH protocol, given in Section 8.

2 Preliminaries

2.1 Zero-Knowledge Arguments

Let \mathcal{R} be a relation generator, such that $\mathcal{R}(1^\lambda)$ returns a polynomial-time decidable binary relation $\mathbf{R} = \{(x, w)\}$, where x is the statement and w is the corresponding witness. We assume one can deduce λ from the description of \mathbf{R} . Let $\mathbf{L}_{\mathbf{R}} = \{x : \exists w \mid (x, w) \in \mathbf{R}\}$ be an **NP**-language containing all the statements which for there exists a corresponding witnesses in \mathbf{R} .

Honest verifier zero-knowledge argument of knowledge. Consider two parties, a prover P and a verifier V . They both have input x , and P additionally has a secret witness w . The prover wants to prove that she knows w for the statement x , such that $(x, w) \in \mathbf{R}$. To accomplish this they carry out an interactive protocol with several rounds, which at the end the verifier gets the transcript (proof) along with x and returns either *accept* or *reject*. Such interactive arguments with 3 rounds are also known as Σ -protocols.

An Honest-Verifier Zero-Knowledge (HVZK) argument Π between (P, V) for the relation \mathbf{R} with knowledge error $\delta(x)$ is supposed to guarantee the following three properties.

Definition 2.1 (Completeness). *An HVZK argument Π with parties (P, V) is perfectly complete for \mathcal{R} , if for $(x, w) \in \mathbf{R}$,*

$$\Pr [\text{trans}(P(\mathbf{R}, x, w) \leftrightarrow V(\mathbf{R}, x)) \text{ is accepted by } V] = 1 ,$$

where trans denotes the transcript of the argument.

Definition 2.2 (Honest Verifier Zero-Knowledge). *The argument Π satisfies honest verifier zero-knowledge for \mathcal{R} , if there exists a PPT algorithm Sim that can simulate the transcript of the argument, such that for all $x \in \mathbf{L}_{\mathbf{R}}$, $(x, w) \in \mathbf{R}$,*

$$\text{trans}(P(\mathbf{R}, x, w) \leftrightarrow V(\mathbf{R}, x)) \approx \text{trans}(\text{Sim}(\mathbf{R}, x) \leftrightarrow V(\mathbf{R}, x))$$

where $\text{trans}(P(\mathbf{R}, x, w) \leftrightarrow V(\mathbf{R}, x))$ shows the transcript of argument Π with (P, V) .

Definition 2.3 (Knowledge Soundness). *The argument Π with parties (P, V) is knowledge sound for \mathcal{R} , if there exists a PPT extraction algorithm Ext , such that for every prover \bar{P} and every $x \in \mathbf{L}_{\mathbf{R}}$, the extraction algorithm Ext satisfies the following condition: let $\gamma(x)$ be the probability that the verifier returns *accept* after interacting with \bar{P} . If $\gamma(x) > \delta(x)$, then upon input $x \in \mathbf{L}_{\mathbf{R}}$, and oracle access to \bar{P} , the algorithm Ext outputs a witness w , s.t. $(x, w) \in \mathbf{R}$ in expected number of steps bounded by $O(\frac{1}{\gamma(x) - \delta(x)})$.*

An HVZK argument of knowledge can be transformed to a Non-Interactive Zero-Knowledge (NIZK) argument of knowledge via the Fiat-Shamir transformation [FS87].

2.2 Commitment Schemes

A commitment scheme allows a committer to commit a secret value, and later open the commitment in a verifiable manner.

Definition 2.4. *A commitment scheme consists of two, possibly randomized, algorithms:*

- An algorithm $\text{Commit}(m)$ which takes a message m and produces a pair (c, o) , where c is the commitment and o is the opening information.
- An algorithm $\text{DeCommit}(c, o)$ which takes the output of Commit and returns m , if (c, o) was not the output of Commit when called with m then the algorithm DeCommit should return \perp .

Two primary requirements for a commitment schemes are known as *hiding* and *binding*, which in summary can be defined as follows.

- **Hiding:** It is hard for any PPT adversary \mathcal{A} , to generate two messages $m_0 \neq m_1$ from the message space \mathcal{M} such that \mathcal{A} can distinguish between their commitments c_0 and c_1 where $(c_0, o_0) = \text{Commit}(m_0)$ and $(c_1, o_1) = \text{Commit}(m_1)$, where o_0 and o_1 are the opening values for the commitments.
- **Binding:** It is hard for any PPT adversary \mathcal{A} , to come up with a collision (c, m_0, o_0, m_1, o_1) , such that (m_0, o_0) and (m_1, o_1) are valid opening values of $m_0 \neq m_1$ for c .

A standard practical commitment scheme is for Commit to generate a high entropy value r and then set $c \leftarrow H(m||r)$ and $o \leftarrow m||r$. If the message m contains sufficient entropy itself, as it does in our application below, one does not need to generate a random string in this construction. Thus the commitment scheme is simply to hash m , and to open we check whether the hash is correct. Such a commitment scheme is both computationally binding (due to collision resistance of the hash function H) and hiding (due to pre-image resistance of the hash function H). We let $\text{Adv}_{\text{Commit}}^{\text{Hiding}}$ (resp. $\text{Adv}_{\text{Commit}}^{\text{Binding}}$) denote the advantage of an adversary in breaking the hiding (resp. binding) property of the commitment scheme.

2.3 Linear Secret Sharing Schemes

Given N parties in the protocol we will denote by P_1, \dots, P_N , we let $\langle x \rangle_q$ denote an additive sharing of a value $x \in \mathbb{F}_q$; i.e. a sharing of x consists of random $x_1, \dots, x_N \in \mathbb{F}_q$ such that $x = \sum_{i \in [N]} x_i$, where P_i holds x_i . We can define the following (linear) operations on shares:

- $\langle x \rangle_q.\text{reveal}()$: To reveal the secret x each party broadcasts its share x_i . Upon receiving x_k from each P_k , P_i sets $x = \sum_{i \in [N]} x_i$.
- $\langle x \rangle_q + \langle y \rangle_q$: Given two sharings, over the same finite field \mathbb{F}_q , each party P_i defines $x_i + y_i$ as its share of the result.
- $\sigma + \langle x \rangle_q$: Given a sharing $\langle x \rangle_q$ and a public constant $\sigma \in \mathbb{F}_q$, party P_1 defines $x_1 + \sigma$ as its new share while the other parties' shares remain the same.
- $\sigma \cdot \langle x \rangle_q$: Given a sharing $\langle x \rangle_q$ and a public constant $\sigma \in \mathbb{F}_q$, each party P_i defines $\sigma \cdot x_i$ as its share of the product.

3 M-Circuits

Given a function F there are many ways of representing the function: theoreticians may look at binary or arithmetic circuit representations, programmers may think of C, Java, or Haskell, a processor designer may think of an x86 instruction stream. By the Church-Turing thesis all are essentially equivalent. In this section we formalize a way of representing a function for use in LSSS-based MPC and/or MPCitH systems. A key aspect of our definition is that the process of compiling/programming an abstract mathematical function F as a concrete representation involves some form of security analysis, i.e. it is not only the MPC/MPCitH protocol which impacts security but also the input representation of the function being operated upon.

3.1 Defining an M-Circuit

At the heart of our definitions is the idea that a function maps input variables to output variables, but that some of the input variables, and indeed some of the output variables may be sensitive.

Machine State. We start by defining a machine state.

Definition 3.1 (Machine State). A machine state *state* defined over a set of finite fields $\mathcal{F} = \{\mathbb{F}_{q_1}, \dots, \mathbb{F}_{q_f}\}$ is a collection of variables (or memory addresses). Each variable has a type which is one of the following three forms:

- $(q_i, -)$, which refers to a variable which holds a non-sensitive variable in the finite field \mathbb{F}_{q_i} ;
- (q_i, s) , which refers to a variable which holds a sensitive variable in the finite field \mathbb{F}_{q_i} ;
- $(-, -)$ which refers to a signed integer variable (i.e. an element of \mathbb{Z}) in some bounded range (for example a 64-bit integer).

The machine state can hold these variables in a number of manners, for example as memory locations indexed by integers via stacks. The usage of the signed integer variables are to allow memory access operations and stack operations within the machine. Note, one could extend the definition to finite rings, and not just finite fields, using techniques such as those from SPDZ2k [CDE+18], but for now we keep to the simpler case of finite fields.

To ease notation in what follows we let $\{x\}_q$ denote a variable of type $(q, -)$, $\langle x \rangle_q$ denote a variable of type (q, s) , and x denote a variable of type $(-, -)$. Also, we make no distinction between the name of the variable and the value it contains. If we want to refer to a type, and are not interested in its sensitivity classification, we refer to the type $(q, *)$, and call it the *base type* of the variable.

We note that variables of the same type can be added, subtracted and multiplied etc. Variables of different types can be combined in the following sense: the operation of a binary arithmetic operator on two variables of type (p_1, s_1) and (p_2, s_2) can be applied if $\gcd(p_1, p_2) \neq 1$, resulting in a type (p_3, s_3) where:

- $p_3 = \gcd(p_1, p_2)$,
- $s_3 = s$ if and only if $s_1 = s$ or $s_2 = s$, otherwise $s_3 = -$. (This means that variables can only become more sensitive, akin to the ‘no write down’ rule of Bell-LaPadula [BL73]).

Thus one can form (relatively) arbitrary arithmetic expressions on variables, and one can assign a type to the result of the expression.

Variables of type $(p, -)$, for prime p , can be arbitrarily converted into variables of type $(-, -)$ and vice-versa, using the inclusion $\mathbb{F}_p \rightarrow [0, \dots, p-1] \subset \mathbb{Z}$ and the mapping $\mathbb{Z} \rightarrow \mathbb{F}_p$ given by $x \mapsto x \pmod{p}$.

Correlated Randomness Sources: As well as variables, and the arithmetic expressions we can create from them, there are two additional components for an M-Circuit, namely *correlated randomness sources* and *gadgets*, which we describe in the following.

Definition 3.2 (Correlated Randomness Source). A correlated randomness source \mathbb{S} is defined by a set of variables $\{v_1, \dots, v_t\}$ of any (specific) types $\{(q_1, *), \dots, (q_t, *)\}$ and a predicate *pred* on those variables.

A correlated randomness source should be thought of as related to the data which is produced in preprocessing phases of MPC protocols such as SPDZ [DPSZ12]. Thus typical sources would be:

- **Triple:** This has associated to it three variables, (a, b, c) all of type (p, s) , for which the predicate is $\text{pred}(a, b, c) := a \cdot b = c$, with a, b being uniformly randomly chosen from \mathbb{F}_p .
- **Square:** This has associated to it two variables, (a, b) , both of type (p, s) for which the predicate is $\text{pred}(a, b) := a \cdot a = b$, with a being uniformly randomly chosen from \mathbb{F}_p .
- **Bit:** This has associated to it a single variable, a , of type (p, s) for which the predicate is $\text{pred}(a) := a \in \{0, 1\}$, and a is uniformly randomly chosen from $\{0, 1\}$.
- **daBit:** This has associated to it two variables, a, b , one of type (p, s) and one of type $(2, s)$, for which the predicate is $\text{pred}(a, b) := (a = b) \wedge (a \in \{0, 1\})$, with a uniformly randomly chosen from $\{0, 1\}$.

Gadgets: The second component we introduce now is called a ‘gadget’. From a high level point of view these can be arbitrary operations. More formally, they are function calls made by the M-Circuit which we do not necessarily implement using an M-Circuit. This means, for example, that their functionality could be provided by some externally defined protocol. In practice, we will use gadgets to perform very specific operations within specific protocols and also to help to define stages of program transformation within a compilation. Thus gadget’s correspond to operations which are done using special protocols, with the idea being that if we can show the special protocol for implementing the gadget is secure and correct, then we can use the gadget as an optimization process within our final protocols.

Definition 3.3 (Gadget). *A gadget \mathbb{G} is a mathematical function which takes a set of variables and outputs a set of variables $(\hat{v}_1, \dots, \hat{v}_u) \leftarrow \mathbb{G}(v_1, \dots, v_t)$, where no assumption is made about how \mathbb{G} will be implemented. The types of the input and output variables are assumed to be implicitly defined by the gadget itself.*

Looking ahead, in the context of MPC using a gadget is like calling a protocol to perform a Garbled Circuit operation on some secret shared data over \mathbb{F}_2 in a system such as that described by the Zaphod paper [AOR⁺19]. The gadget in this case goes outside the neat confines of LSSS-based MPC, but it is integrated with the LSSS based MPC and is thus able to allow greater functionality at reduced cost. Another example of a gadget could be a multiplication gate, which we do not expand into its Beaver representation if we want to avoid correlated randomness sources.

Instructions and M-Circuits. An M-Circuit is composed of an *ordered* finite list of instructions as follows.

Definition 3.4 (Instruction). *An instruction can be one of the following forms:*

- *A pair (v, expr) , where v is a variable and expr , is an arithmetic expression as described above. The type of v must correspond to the type of expr . As a shorthand we may write $v \leftarrow \text{expr}$. We restrict the expressions expr to be arbitrary arithmetic expressions, however the total degree of the expression in any sensitive variables must be one. Thus we can only compute linear functions on sensitive variables.*
- *A tuple $(\{v_1, \dots, v_t\}, \mathbb{S})$ where \mathbb{S} is a correlated randomness source, and the variables $\{v_1, \dots, v_t\}$ have the same types as the variables associated to the source. As a shorthand we may write $v_1, \dots, v_t \leftarrow \mathbb{S}$.*

- A tuple $(\{\hat{v}_1, \dots, \hat{v}_u\}, \{v_1, \dots, v_t\}, \mathbb{G})$ where \hat{v}_i and v_i are variables and \mathbb{G} is a gadget as described above. The types of v_j and \hat{v}_i must correspond to the input and output types of the Gadget. As a shorthand we may write $\hat{v}_1, \dots, \hat{v}_u \leftarrow \mathbb{G}(v_1, \dots, v_t)$.
- A ‘declassification’ instruction which we write as $x \leftarrow y.\text{reveal}()$. This takes a variable y of type (p, s) and creates a variable x of type $(p, -)$ which has the same value as y .
- A special instruction called **terminate**.

Examples, of the first three types of instruction, could include:

$$\begin{aligned} \langle z \rangle &\leftarrow \langle x \rangle_p + \langle y \rangle_p \\ \langle x \rangle_p, \langle y \rangle_p, \langle z \rangle_p &\leftarrow \text{Triple} \\ \{\langle c_i \rangle_2\}_{i=0}^{127} &\leftarrow \text{AES} \left(\{\langle k_i \rangle_2\}_{i=0}^{127}, \{\langle m_i \rangle_2\}_{i=0}^{127} \right). \end{aligned}$$

Finally, we can define what we mean by an M-Circuit.

Definition 3.5. An M-Circuit is a tuple consisting of an ordered list of instructions \mathcal{I} and two sets of variables \mathcal{V}_I and \mathcal{V}_O (called the input and the output variables).

A class of M-Circuits $\mathcal{C}_{\{\mathbb{F}_{q_1}, \dots, \mathbb{F}_{q_f}\}}^{\{\mathbb{S}_1, \dots, \mathbb{S}_s\}, \{\mathbb{G}_1, \dots, \mathbb{G}_g\}}$ is the set of all M-Circuits over the finite fields $\{\mathbb{F}_{q_1}, \dots, \mathbb{F}_{q_f}\}$, which utilize correlated randomness sources $\{\mathbb{S}_1, \dots, \mathbb{S}_s\}$ and gadgets $\{\mathbb{G}_1, \dots, \mathbb{G}_g\}$. If \mathcal{F} and \mathcal{F}' are sets of finite fields, and \mathcal{S} and \mathcal{S}' are sets of correlated randomness sources, and \mathcal{G} and \mathcal{G}' are sets of gadgets then we have $\mathcal{C}_{\mathcal{F}}^{(\mathcal{S}, \mathcal{G})} \subseteq \mathcal{C}_{\mathcal{F}'}^{(\mathcal{S}', \mathcal{G}')}$ if $\mathcal{F} \subseteq \mathcal{F}'$, $\mathcal{S} \subseteq \mathcal{S}'$ and $\mathcal{G} \subseteq \mathcal{G}'$.

3.2 Executing an M-Circuit

An M-Circuit $(\mathcal{I}, \mathcal{V}_I, \mathcal{V}_O)$ operates on a machine state as follows. The machine state **state** has an initial state consisting of the set of registers \mathcal{V}_I with pre-assigned values given to them (i.e. the inputs to the function). In addition there is a special register of type $(-, -)$, called **pc**, which is initial set to zero. Then the following operations are repeated until a **terminate** instruction is met.

- Instruction numbered **pc** is fetched from the list of instructions \mathcal{I} .
- The value of **pc** is incremented by one.
- The instruction is executed as follows depending on its type
 - (v, expr) is evaluated if all the variables in **expr** are currently defined, and the result is assigned to variable v . If not all variables are defined then the system aborts.
 - $(\{v_1, \dots, v_t\}, \mathbb{S})$ is evaluated by sampling the variables $\{v_1, \dots, v_t\}$ according to the source definition.
 - $(\{\hat{v}_1, \dots, \hat{v}_u\}, \{v_1, \dots, v_t\}, \mathbb{G})$ is evaluated as for (v, expr)
 - Declassification instructions do the obvious declassification operation.
 - If the instruction is **terminate** then the M-Circuit terminates.

On termination the M-Circuit outputs the variables in the set \mathcal{V}_O , if they are defined. If any are not defined it aborts.

Note, this is a rather general model in a number of senses:

- One can perform a conditional branch on non-sensitive variables by making instructions of the form (pc, expr) , e.g. $\text{pc} \leftarrow b \cdot 100 + (1 - b) \cdot 200$ will either result in a jump to instruction 100 or instruction 200 depending on the value of variable $b \in \{0, 1\}$ of type $(-, -)$.

- Subroutines calls, and hence recursion, can be performed by using creating a stack of type $(-, -)$ in the machine state, and then using this to push/pop the pc variable on or off of it.

The main limitation of the model seems to be that instructions of the form (v, expr) can only contain linear functions of sensitive variables. This is where our gadgets and randomness sources will come in.

There are four different measures of complexity of an M-Circuit, and we name these so as to link them with their analogues when we use M-Circuits for MPC (where analogues exist), as follows.

- The *offline complexity* is the number of calls to the source oracles $\{\mathbb{S}_1, \dots, \mathbb{S}_s\}$ made by the M-Circuit on a given input.
- The *online communication complexity* is the number of calls to the operation $\text{reveal}()$ made by the M-Circuit on a given input.
- The *online round complexity* is the minimum number of *parallel* calls to the operation $\text{reveal}()$ made by the M-Circuit on a given input.
- The *gadget-complexity* (which has no usual analogue in the MPC domain) is the number of calls to gadgets \mathbb{G} made by the M-Circuit on a given input.

3.3 Compiling M-Circuits

An M-Circuit is created by a process called compilation.

Definition 3.6 (Compilation). *A compilation step is an algorithm which takes an M-Circuit C in a class $\mathcal{C}_{\mathcal{F}}^{(S, \mathcal{G})}$ and maps it to an M-Circuit C' in a class $\mathcal{C}_{\mathcal{F}'}^{(S', \mathcal{G}')}$. The algorithm must ensure that the functional behaviour of C and C' are identical, i.e. the input/output behaviour of C and C' are the same.*

Note a compilation says nothing about whether $\mathcal{C}_{\mathcal{F}}^{(S, \mathcal{G})} \subseteq \mathcal{C}_{\mathcal{F}'}^{(S', \mathcal{G}')}$ or vice-versa.

Given an arbitrary polynomial time function F , defined over a set of finite fields \mathcal{F} and the integers, there is always an M-Circuit, which we call C_F , which implements F in the class $\mathcal{C}_{\mathcal{F}}^{(\emptyset, \{F\})}$, namely the M-Circuit which uses the gadget $\mathbb{G} = F$. The goal of compilation is to find representations of C_F in simpler classes, in particular a class which can be implemented in either an MPC or MPCitH system. We present three exemplar compilations here to fix ideas:

- **Arithmetic Circuit:** Consider the gadget \mathbb{G}_M for a finite field \mathbb{F}_p which multiplies two input values, giving the output value of the same type. Then the standard ‘arithmetization’ of polynomial time functions F compiles the M-Circuit C_F to an M-Circuit C_F^A in the class $\mathcal{C}_{\mathbb{F}_p}^{(\emptyset, \{\mathbb{G}_M\})}$.
- **Beaver Randomized Circuit:** We can take the M-Circuit C_F^A produces in the previous example and compile it to an M-Circuit C_F^B in the class $\mathcal{C}_{\mathbb{F}_p}^{(\text{Triple}, \{\emptyset\})}$ using Beaver’s standard circuit randomization trick, [Bea92].
- **Insecure Circuit:** Here we take F , and create the functionally equivalent function F' which first de-classifies all the sensitive input variables of F . Then it evaluates F on the clear values, using the arithmetic circuit representation of F . Finally, it re-classifies any output variables as sensitive which need to be sensitive, by multiplication by $\langle 1 \rangle_p$. The associated arithmetic circuit $C_{F'}^A$ (which includes $\text{reveal}()$ operations as well as the usual arithmetic operations) is an M-Circuit in the class $\mathcal{C}_{\mathbb{F}_p}^{(\emptyset, \{\mathbb{G}_M\})}$.

The last example here hints that compilation can create something which is ‘insecure’. To quantify this notion we need to define what we mean by security of an M-Circuit.

3.4 Security of M-Circuits

To define security of an M-Circuit we have to examine the `reveal()` operations in more detail, since these are the operations which potentially de-classify sensitive information. Informally we require that the `reveal()` operations never reveal more than a negligible amount of sensitive information about any sensitive inputs to the function.

To each `reveal` operation of the form $a \leftarrow \langle b \rangle_q.\text{reveal}()$ we associate a given distribution \mathcal{R}_b on the set \mathbb{F}_q . The reader can think of \mathcal{R}_b on first reading as the uniform distribution (which will be true for circuits compiled using the Beaver compilation above, but it is not true in general). To take into account the type of efficient function representations used in say [CS10] we need to be a little more nuanced.

The distributions \mathcal{R}_b are on the outputs of `reveal()` are conditioned on the following three things:

- The specific non-sensitive inputs and outputs of the function being evaluated.
- The random execution path taken by the circuit.
- The distributions of the correlated randomness sources.

However, the distributions are not conditioned on sensitive input and output values to the function. For example consider the code fragments in Figure 1. In fragment (a) the function is b , in which case the distribution \mathcal{R}_z has probability mass of one at the value $b - 1$ and is zero elsewhere, whilst in fragment (b) the distribution \mathcal{R}_z is the set of values in the range $[-B, \dots, B]$ with an associated binomial distribution.

<p>Code Fragment (a)</p> <pre> a = z.reveal() b = a+1 Output b </pre>	<p>Code Fragment (b)</p> <pre> for i in range(2*B): b_i = Bits z = sum(b[2*i]-b[2*i+1],i in range(B)) a = z.reveal() </pre>
<p>Code Fragment (c)</p> <pre> z=x+y a=z.reveal() Output a </pre>	<p>Code Fragment (d)</p> <pre> b=x.reveal() c=y.reveal() a=b+c Output a </pre>

Fig. 1. Example Code Fragments

The trace Trace_C of an M-Circuit, on a given input, consists of the set of non-sensitive input variables, the non-sensitive output variables, plus the output of every `reveal()` operation. A simulated trace Sim_C is the same except that the output of every `reveal()` operation is replaced by a value chosen via the distributions \mathcal{R}_b above.

Definition 3.7 (Perfectly Secure M-Circuit). *An M-Circuit C is said to perfectly securely implement a function F if the functional behaviour of the M-Circuit C and the M-Circuit C_F are identical, and the distribution of Trace_C and Sim_C are identical for all input values.*

Definition 3.8 (Statistically Secure M-Circuit). *The M-Circuit C is said to securely implement a function F with statistical security sec if the functional behaviour of the M-Circuit C and the function C_F are identical, and the statistical distance between the distribution of Trace_C and Sim_C is bounded by $2^{-\text{sec}}$ for all input values.*

The first question one must ask is if such a definition is vacuous. The celebrated technique of Beaver’s Circuit Randomization [Bea92] says no.

Theorem 3.1. *Every polynomial time function F can be perfectly securely implemented by the M-Circuit C_F^B with polynomial complexity (in all four metrics).*

Proof. Using the compilation process above we can compile the M-Circuit C_F^B . It is well known that the `reveal()` operations this creates are associated with uniform distributions, and thus the reveals are perfectly hiding.

Note, this definition is about the representation of the function i.e. the compilation of the M-Circuit from the function definition. It asks whether the compilation process is itself secure; it makes no claim about how the M-Circuit is then used or evaluated within an MPC or MPCitH system.

Not all compilations will result in secure M-Circuits, as our insecure compilation example illustrates. To see why this compilation violates our security definition, consider the specific function F which takes two sensitive values x and y , and returns their sum, but as a non-sensitive value. Mathematically one could write $F(\langle x \rangle, \langle y \rangle) = (\langle x \rangle + \langle y \rangle).reveal()$. A functionally valid M-Circuit for this function is given in code fragment (c) of Figure 1, whilst another functionally valid M-Circuit for the same function is given in code fragment (d).

Code fragment (c) is a perfectly secure M-Circuit, with the distribution \mathcal{R}_z being the point distribution will all the probability mass at the point a (where a is the public output of the function). Thus the valid transcript Trace_C and Sim_C are identical and equal to $\text{Trace}_C = \text{Sim}_C = \{\emptyset, \{a\}, \{a\}\}$, i.e. there are no distributions here at all, Trace_C and Sim_C are fixed by the output a . Note, the \emptyset corresponds to the set of non-sensitive input variables, the first $\{a\}$ is the set of non-sensitive output variables, and the second $\{a\}$ is the output of every reveal (resp. the simulated reveals) in the case of the actual trace (resp. the simulated trace).

Code fragment (d) has \mathcal{R}_x being the point distribution on x , with y being the point distribution of $y = a - x$. Thus Trace_C of the second M-Circuit is equal to $\text{Trace}_C = \{\emptyset, \{a\}, \{x, a - x\}\}$ which is a fixed value (for each given input), whereas the simulated trace \mathcal{S}_C is equal to the value given by $\mathcal{S}_C = \{\emptyset, \{a\}, \{r, a - r\}\}$ where r uniformly chosen from \mathbb{F}_p . Thus Trace_C and \mathcal{S}_C in the second case can never be statistically close. Thus the second compilation to an M-Circuit is insecure, but functionally correct.

4 M-Circuits for Multi-Party Computation

It turns out that our M-Circuit notion lies underneath almost all algorithmic level optimizations of LSSS-based MPC over the last decade (by which we mean optimizations related to the program representation and not the MPC protocol itself). The M-Circuit concept allows us to isolate which optimizations can be utilized by which MPC protocols, since not all MPC protocols can implement all M-Circuit classes.

As remarked earlier, arithmetic circuit representation of a functionality over a finite field \mathbb{F}_q correspond to M-Circuits in the class $\mathcal{C}_{\mathbb{F}_q}^{(\emptyset, \{G_M\})}$. Thus ‘traditional’ LSSS based MPC protocols such as [BGW88, CCD88], or modern protocols such as [CGH⁺18], which have specific protocols for the multiplication operation can utilize this M-Circuit representation. However, the security of these protocols is then proved by showing that the implementation of the specific multiplication gadget leaks no information.

Protocols which expand the multiplication gadget via Beaver’s trick [Bea92] utilize circuits from the class $\mathcal{C}_{\mathbb{F}_p}^{(\{\text{Triple}\}, \emptyset)}$. The security of the underlying (passively secure) online protocol then follows from the security of the M-Circuit representation; if the M-Circuit is secure then so is the obvious LSSS-based MPC protocol in which one replaces the sensitive variables in the M-Circuit by secret shared values. The problem comes in creating an offline phase to produce the necessary correlated randomness source `Triple` in a secret shared manner. For honest majority protocols this offline phase is usually performed using hyper-invertible matrices (as in VIFF [DGKN09]) or, for dishonest majority protocols using homomorphic encryption (as in SPDZ [DPSZ12]) or OT (as in MASCOT [KOS16]). In the latter case to prove active security of the underlying MPC protocol one needs to provide a form of authenticated secret sharing, while the privacy of the protocol follows from the security of the M-Circuit representation. For the passive case the security of the online phase we cover in Section 5.1 later.

Papers such as [DFK⁺06, CS10, Cd10] showed that one can obtain greater efficiency by working with M-Circuits in the class $\mathcal{C}_{\mathbb{F}_{2^8}, \mathbb{F}_q}^{(\emptyset, \{\mathbb{G}_M\})}$, or equivalently $\mathcal{C}_{\mathbb{F}_{2^8}, \mathbb{F}_q}^{(\{\text{Triple}_{2^8}, \text{Triple}_q\}, \emptyset)}$; although of course they did not use this language. In these latter works the authors used multiplication to create shared-random bits, whereas if one assumes these as a random source then the protocols become simpler to describe; thus the same work can be cast as corresponding to M-Circuits in the class $\mathcal{C}_{\mathbb{F}_q}^{(\{\text{Triple}, \text{Bit}\}, \emptyset)}$. It is this latter representation which is used in modern LSSS-based systems in the pre-processing model; for example the second generation of the SPDZ protocol [DKL⁺13] utilizes function descriptions which are M-Circuits in the class $\mathcal{C}_{\mathbb{F}_p}^{(\{\text{Triple}, \text{Square}, \text{Bit}\}, \emptyset)}$. The papers such as [CS10, Cd10] also showed one can obtain more efficient representations, in terms of minimizing the various complexity measures we described earlier, by compiling to what we call statistically secure M-Circuits as opposed to perfectly secure M-Circuits.

Systems which make use of daBits [RW19] to translate between binary and arithmetic fields utilize M-Circuits in the class $\mathcal{C}_{\{\mathbb{F}_p, \mathbb{F}_2\}}^{(\{\text{Triple}, \text{Square}, \text{Bit}, \text{daBit}\}, \emptyset)}$. Systems such as Zaphod, [AOR⁺19] extend this idea further by allowing gadgets based on garbled circuits to be evaluated within the MPC-computation. Thus they allow M-Circuits in the class $\mathcal{C}_{\{\mathbb{F}_p, \mathbb{F}_2\}}^{(\{\text{Triple}, \text{Square}, \text{Bit}, \text{daBit}\}, \{\mathbb{G}_1, \dots, \mathbb{G}_g\})}$ for specific garbled circuit based sub-procedures $\mathbb{G}_1, \dots, \mathbb{G}_g$. As long as the gadget can be securely implemented, then the overall MPC protocol is itself secure.

Obviously compilation methods, and different sources of correlated randomness, will give different M-Circuits with different complexities. This is essentially the engineering challenge of MPC solutions: to pick the compilation strategy and sources of correlated randomness in order to achieve an efficient M-Circuit which can be executed by a given MPC engine.

5 M-Circuits for MPC-in-the-Head

In this section, we present an MPCitH-based Honest Verifier Zero-Knowledge (HVZK) argument of knowledge system for satisfiability of a function (computation) that is compiled to an M-Circuit. Initially we consider M-Circuits with no gadgets, namely we consider the class of M-Circuits $\mathcal{C}_{\{\mathbb{F}_{q_1}, \dots, \mathbb{F}_{q_f}\}}^{(\{\mathbb{S}_1, \dots, \mathbb{S}_s\}, \emptyset)}$. Later we shall remove this restriction, at the expense of introducing more rounds of communication.

Our construction extends Katz et al.’s construction [KKW18] for arbitrary finite fields. Since we work with M-Circuits, we do not consider operations such as AND, multiplication or squaring (as in Katz et al.’s presentation), but rather we formalize the protocol in term of generic correlated

randomness sources over arbitrary finite fields, along with calls to the `reveal()` function. As we have already explained, such a representation is universal, and can lead to optimizations (which we will discuss later).

We first describe the specific underlying MPC protocol to securely compute an M-Circuit instance that we will exploit in our MPCitH protocol. Then, we present an MPCitH-based HVZK argument of knowledge based on the input M-Circuit instance. Initially, we present a protocol which checks the correlated randomness sources using the *cut-and-choose* paradigm. This method works for arbitrary sources. In a latter section we present another methodology which works for some specific correlated randomness sources which is based on the *sacrificing* idea used in some actively secure MPC protocols.

5.1 The Underlying MPC Protocol

The MPCitH protocol we utilize will make use of a very simple (passively secure) MPC protocol based on full threshold secret sharing in the pre-processing model. The function F we will be evaluating is assumed to have (some) sensitive input variables, but no sensitive output variables. The N parties in the protocol we will denote by P_1, \dots, P_N .

Offline Phase. We define an ideal functionality for the offline phase, which implements the generation of suitable correlated randomness according to the sources required by the M-Circuit. This is given in Figure 2.

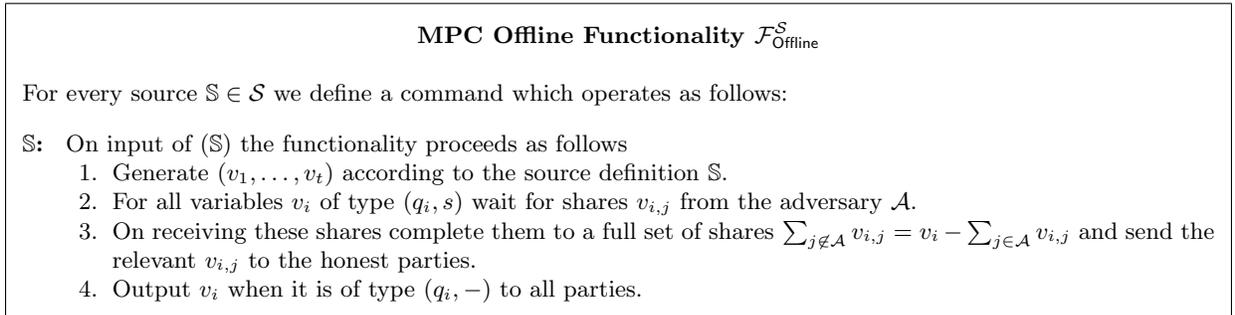


Figure 2. Functionality $\mathcal{F}_{\text{Offline}}^{\mathcal{S}}$

Online Phase. We wish to implement the passively MPC/SFE functionality given in Figure 3, in which we assume the sensitive inputs are assigned to specific parties. This is done using the online phase given in Figure 4, which is defined in the $\mathcal{F}_{\text{Offline}}$ -hybrid model.

Security. We let \mathcal{A} denote an adversary which statically corrupts a subset of the parties. We abuse notation slightly by referring to \mathcal{A} both as the adversary, and as the set of parties which it has corrupted. We define $\text{view}^{\{\mathcal{A}, \Pi_{\text{MPC}}\}}(C)$ to be the view of \mathcal{A} during the execution of the protocol Π_{MPC} on the function F represented by the M-Circuit C in the $\mathcal{F}_{\text{Offline}}$ -hybrid model. This view consists of the inputs of the parties in \mathcal{A} , the shares of the correlated randomness the parties in \mathcal{A} receive from $\{\mathbb{S}_1, \dots, \mathbb{S}_s\}$, and the messages they obtain from the other parties while evaluating the protocol. The security of Π is stated in the following theorem.

Passively Secure MPC/SFE Functionality \mathcal{F}_{MPC}

Given a function F defined over finite fields with input variables V_I and output variables V_O , the functionality proceeds as follows.

1. For each input variable $v \in V_I$:
 - (a) If input variable v of type (q, s) is assigned to party P_i then wait for party P_i to enter the value v .
 - (b) If input variable v is of type $(q, -)$ then wait for all parties to input the same value v .
2. Compute the function F on (v_1, \dots, v_t) and output the output variables to all parties (recall we assume F has no sensitive output variables).

Figure 3. Passively Secure MPC/SFE Functionality \mathcal{F}_{MPC}

Theorem 5.1. *For every subset of parties $\mathcal{A} \subseteq \{P_1, \dots, P_N\}$, with $|\mathcal{A}| \leq N - 1$, there exists a probabilistic polynomial-time algorithm \mathcal{S} , with access to the functionality \mathcal{F}_{MPC} , such that $\{\mathcal{S}(\mathcal{A}, F, v_{\mathcal{A}})\} \equiv \text{view}^{\{\mathcal{A}, \Pi_{\text{MPC}}\}}(C)$, where $v_{\mathcal{A}}$ are the function inputs of the parties in \mathcal{A} .*

The equivalence relation is a perfect equivalence if the M-Circuit C is a perfectly secure implementation of the functionality F , and is a statistical equivalence if the M-Circuit is a statistically secure implementation of F .

Proof. Intuitively, the corrupted parties see only shares of the values on the variables of the M-Circuit that could reveal to any value or random public values. The simulator has to just execute the M-Circuit, and add the necessary data to the adversaries view. Technically the algorithm \mathcal{S} acts as follows,

- It first chooses random shares for the parties in \mathcal{A} , for each of the input variables of the parties not in \mathcal{A} , and adds them to the view of the corrupted parties \mathcal{A} .
- It obtains the inputs for the sensitive variables assigned to the parties in \mathcal{A} from \mathcal{A} , and then calls F (using random values for the honest parties inputs) so as to obtain the output values of F .
- Then the simulator passes over the M-Circuit step by step and performs all the local operations on the corrupted parties' shares based on the protocol,
- For each query to a correlated randomness sources, it chooses the shares of the sensitive outputs of the sources at random and adds them to the corrupted parties' view, the non-sensitive outputs are added directly to the view.
- For each $\text{reveal}()$ call, say $\langle b \rangle_q.\text{reveal}()$, the simulator chooses α from the corresponding distribution \mathcal{R}_b on the set \mathbb{F}_q . Note, the simulator knows the distributions \mathcal{R}_b since it knows the program, the non-sensitive inputs, the sensitive input values of the parties in \mathcal{A} and the output of the function F . It then takes the corrupted parties shares b_i for $i \in \mathcal{A}$ and defines the honest parties shares by $\sum_{i \notin \mathcal{A}} b_i = \alpha - \sum_{i \in \mathcal{A}} b_i$. The values (b_1, \dots, b_N) are added to the view.

The only difference between the real protocol view and the simulated view is in dealing with the $\text{reveal}()$ operations. That these have the required difference between the real and simulated view follows from the security definition of the underlying M-Circuit M . □

5.2 Sub-Procedures for MPCitH

In this subsection we collect together a number of sub-procedures and algorithms for our general MPCitH protocol for M-Circuits.

Passively Secure MPC/SFE Protocol $\Pi_{\text{MPC}}^{(\mathcal{F}, \mathcal{S})}(C)$

Given a function F defined over finite fields with input variables V_I and output variables V_O represented as an M-Circuit C in the class $\mathcal{C}_{\mathcal{F}}^{(\mathcal{S}, \emptyset)}$, the protocol proceeds as follows.

1. For each input variable $v \in V_I$
 - (a) If input variable v of type (q, s) is assigned to party P_i then party P_i shares $v = \sum v_j$ and sends v_j to party P_j .
 - (b) If input variable v is of type $(q, -)$ then all parties agree on the value v .
2. Now execute the M-Circuit line by line (as above).
 - For a $x.\text{reveal}()$ command, party P_i sends his share x_i to all parties.
 - For a call to a correlated randomness source $\mathbb{S} \in \mathcal{S}$, make the appropriate call to the functionality $\mathcal{F}_{\text{Offline}}$.
 - For an arithmetic operation, perform the associated operation on the linear secret sharing scheme given above.
3. Finally, for a terminate operation, for each variable $v \in V_O$ the parties output their (necessarily opened) value v as their output.

Figure 4. Passively Secure MPC/SFE Protocol $\Pi_{\text{MPC}}^{(\mathcal{F}, \mathcal{S})}(C)$

Pseudo-Random Generator. We let PRG_q denote a pseudo-random function, which on input of a key seed and an index j outputs a (pseudo-) uniformly random element of the finite field \mathbb{F}_q . We let PRF_λ denote an equivalent function which outputs values in $\{0, 1\}^\lambda$.

GenAux Function. To a correlated randomness source \mathbb{S} we associate a deterministic algorithm $\text{GenAux}^{\mathbb{S}}$ which on input of a given assignment to the variables $\{v_1, \dots, v_t\}$ in the source will output a set of variables $\{v'_1, \dots, v'_t\}$ of the same types. The output should satisfy the following equality of distributions, where $v_1 \leftarrow \mathbb{F}_{p_1}$ means sample v_1 uniformly from the field \mathbb{F}_{p_1} ,

$$\begin{aligned} & \left\{ (v_1 + v'_1, \dots, v_t + v'_t) : v_i \leftarrow \mathbb{F}_{p_i} \text{ for } i = 1, \dots, t, \right. \\ & \quad \left. (v'_1, \dots, v'_t) \leftarrow \text{GenAux}^{\mathbb{S}}(v_1, \dots, v_t) \right\} \\ & \equiv \left\{ (v_1, \dots, v_t) : (v_1, \dots, v_t) \leftarrow \mathbb{S} \right\} \end{aligned}$$

Note, there can be many ways for a given source to define the algorithm GenAux ; some are more compact than others. For example take the source **Triple** which has (at least) the two following definitions for GenAux .

1. $\text{GenAux}^{\text{Triple}}(a, b, c) = (0, 0, a \cdot b - c)$.
2. $\text{GenAux}^{\text{Triple}}(a, b, c) = (x - a, y - b, z - c)$ where x, y are deterministically selected from \mathbb{F}_p by $\text{GenAux}^{\text{Triple}}$ using a PRG with the seed $H(a, b, c)$, for some hash function H , and $z = x \cdot y$.

The first of these is more efficient in our application as the user knows the first two coordinates are always zero, and can therefore drop them from any data transferred. It turns out the first is also better for one of our optimizations we present later.

Sources which require some specific distribution, such as the **Bit** source from earlier, can be produced by defining $\text{GenAux}^{\text{Bit}}(a) = (a - b)$ where $b = H(a) \& 1$ for some hash function H .

GenShares. To each correlated randomness source \mathbb{S} , with variables $\{v_1, \dots, v_t\}$ we associate the following seeds:

1. If v_i is of type $(q_i, -)$ then we associate a single seed $\text{seed}_i^{\mathbb{S}}$.

2. If v_i is of type (q_i, s) then we associate N seeds $\text{seed}_{i,j}^{\mathbb{S}}$ for $j = 1, \dots, N$.

We also associate a counter $\text{cnt}^{\mathbb{S}}$, which on initialization of the source is set to zero. In the MPCitH protocol below when \mathbb{S} is called we execute an algorithm $\text{GenShares}^{\mathbb{S}}$ (given in Figure 5) which takes as input the above seeds and the counter $\text{cnt}^{\mathbb{S}}$ and produces a sample from the randomness source presented as a sharing amongst the parties, as well as the correction term. We write $(\{v_i\}_{\dagger}, \{v_{i,j}\}_*, \text{aux}, \text{cnt}^{\mathbb{S}}) \leftarrow \text{GenShares}^{\mathbb{S}}(\{\text{seed}_i^{\mathbb{S}}\}_{\dagger}, \{\text{seed}_{i,j}^{\mathbb{S}}\}_*, \text{cnt}^{\mathbb{S}})$.

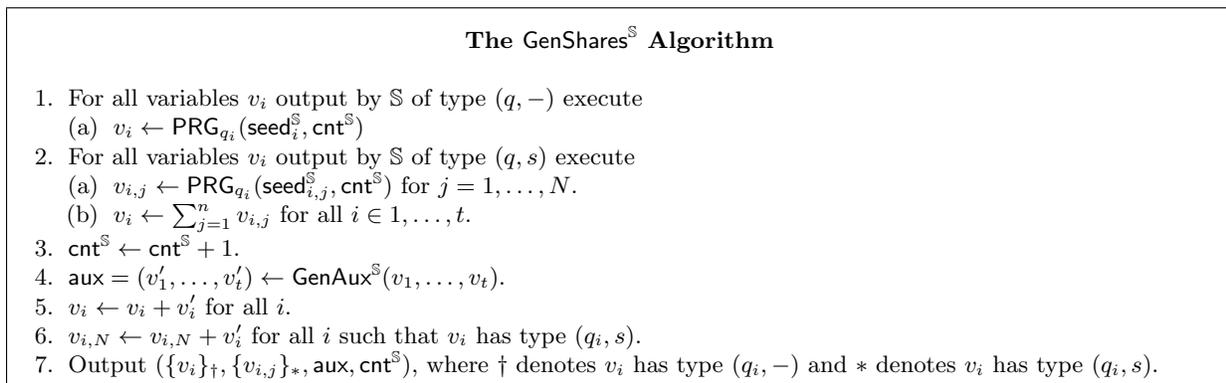


Figure 5. The $\text{GenShares}^{\mathbb{S}}$ algorithm for a source \mathbb{S}

5.3 The Construction of HVZK Argument of Knowledge

We can now present our generalization of the MPCitH protocols of [KKW18, BN20] to the case of arbitrary M-Circuits. Our initial construction uses the *cut-and-choose* checking paradigm, but later we will also consider the other checking approach, i.e. sacrificing, that we show to be more efficient for particular cases. Recall at this point we assume an M-Circuit C is given in the class $\mathcal{C}_{\{\mathbb{F}_{q_1}, \dots, \mathbb{F}_{q_f}\}}^{\{\{\mathbb{S}_1, \dots, \mathbb{S}_s\}, \emptyset\}}$. The M-Circuit has no sensitive output variables, but there are a set of sensitive input variables, which we denote by \mathbf{w} . The prover wishes to show that he knows a witness for these output variables, which by abuse of notation we also call \mathbf{w} , such that the M-Circuit produces a given output.

At a high level the proof proceeds by the prover simulating the N -party MPC protocol from Section 5.1 in his head and executes it over an additive sharing of \mathbf{w} , along with calls to the correlated randomness sources $(\{\mathbb{S}_1, \dots, \mathbb{S}_s\}, \emptyset)$, which are performed using the algorithm $\text{GenShares}^{\mathbb{S}}$ given above. Clearly, as the secret sharing scheme is executed in prover's head, the prover might try to cheat and convince the verifier about a false statement. To prevent such issues, and hence to obtain a negligible soundness error, the construction allows the verifier to challenge the prover. Namely, at the end of the first round the prover commits (by the above commitment scheme) to the views of N parties in M executions. Then, the verifier (randomly) challenges a subset of executions $E \subset [M]$ of size τ for which all correction terms induced by calls to the correlated randomness sources will be revealed and verified. The verifier also (randomly) challenges a single party $j \in [N]$, such that all parties views are opened to him (bar party j) for all $e \in [M] \setminus E$.

After revealing the secret information for the challenged executions and parties, e.g. the master seeds, the challenged parties' seeds, or the commitments, the verifier recomputes (either using

directly the values sent by the Prover, or by using the parties' seeds and correction terms to emulate the secret sharing scheme) and checks the commitments and final output of the M-Circuit.

In the described HVZK argument, intuitively, zero-knowledge is achieved relying on the fact that the M-Circuit is secure (its trace is simulatable) and the revealed data are only random values which are independent of the witness \mathbf{w} . Thus the $N - 1$ views that are revealed do not reveal anything as the underlying MPC protocol is passively secure against $N - 1$ semi-honest parties.

In our protocol description, as before, we use $\langle x \rangle_q$ to denote a sensitive variable (associated to an additive sharing $x = \sum_{j \in [N]} x_j$) and $\{x\}_q$ to denote a non-sensitive variable within the M-Circuit. As input to both the protocol and the verifier we have a general M-circuit C in the class $\mathcal{C}_{\mathcal{F}}^{(\mathcal{S}, \emptyset)}$, with the set of finite fields $\mathcal{F} = \{\mathbb{F}_{q_1}, \dots, \mathbb{F}_{q_f}\}$, a set of Correlated Randomness Sources $\mathcal{S} = \{\mathbb{S}_1, \dots, \mathbb{S}_s\}$, and no gadget. We assume that the prover and verifier have agreed on the non-sensitive input variables to the M-Circuit, and the prover additionally has an assignment to the *sensitive* input variables (witness) such that the M-Circuit evaluates to a given public output.

Phase 1 of the protocol is given in Figure 6, at the end of this execution the prover sends to the verifier the triple $(h_\phi, h_\Gamma, h_{\text{view}})$. Phase 2, given in Figure 7, starts with the verifier (randomly) selecting a subset of executions $E \subset [M]$ of size τ . We set $\bar{E} = [M] \setminus E$, and the verifier also selects $j_e \in [N]$ for $e \in \bar{E}$. To ease notation we define $I_e = [N] \setminus j_e$. The challenge is the set E and the values $\{j_e\}_{e \in \bar{E}}$. Figure Figure 7 also gives, in Phase 3, what the verifier finally performs to verify the proof is correct.

Remark. Note that the M-Circuits can have a randomized execution even for a deterministic function. Therefore the prover computes *all* the M executions in Phase 1, even those for which the master seed will be revealed. This is needed to *learn* the number of calls to the correlated random sources in each execution.

5.4 Security Proof

Next, we show that the construction of an MPCitH protocol, given in Figure 6 and Figure 7, is an HVZK argument of knowledge which works for all functions represented as an M-Circuit in the class $\mathcal{C}_{\{\mathbb{F}_{q_1}, \dots, \mathbb{F}_{q_f}\}}^{\{\{\mathbb{S}_1, \dots, \mathbb{S}_s\}, \emptyset\}}$. Since our protocol extends previous MPCitH-based constructions, such as [KKW18, BN20], our security proofs are an adaption of their proofs for our setting.

Let c be the number of executions where the prover cheats in the protocol, by for example producing invalid values of the output of the function $\text{GenShares}^{\mathbb{S}}$. We let A denote the event that among the M total pre-processing runs, none of the τ that are opened coincide with the c ones that the adversary chose to cheat in. Since τ emulations out of M are revealed and checked by the verifier (in Step 1 of Phase 3), the success probability of an adversarial prover in passing the checks on the correlated randomness is $\Pr[A_c] = \frac{\binom{M-c}{\tau}}{\binom{M}{\tau}}$.

After the correlated randomness is verified in Step 2 of Phase 3, the remaining $M - \tau$ M-Circuit executions are then verified by the prover opening all but one players view. In order to make the output proof accept, the prover must cheat (i.e., deviate from the specification of the secret sharing scheme) in $M - \tau - c$ emulations. We let B denote the event that for the $M - \tau - c$ online executions with honest pre-processing the dishonest party is not opened. Since $N - 1$ views are being opened in each such emulation, the prover clearly cannot cheat in the view of more than one party. Therefore, his success probability in this stage is $\Pr[B_c] = 1/(N^{M-\tau-c})$.

Phase 1 of the MPCitH protocol

Phase 1:

For $e \in [M]$ the prover executes

Set up the seeds

1. $\text{seed}_e \leftarrow \{0, 1\}^\lambda$.
2. For $i \in [N]$ compute $\text{seed}_{e,i} \leftarrow \text{PRF}_\lambda(\text{seed}_e, i)$.
3. For $j \in [s]$ and variable v_k in source \mathbb{S}_j
 - $\text{seed}_{e,k}^{\mathbb{S}_j} \leftarrow \text{PRF}_\lambda(\text{seed}_e, j \| k)$ if v_k is of type $(q_i, -)$.
 - $\text{seed}_{e,i,k}^{\mathbb{S}_j} \leftarrow \text{PRF}_\lambda(\text{seed}_e, j \| k)$ if v_k is of type (q_i, s) for $i \in [N]$.
4. For $j \in [s]$ set $\text{cnt}_e^{\mathbb{S}_j} \leftarrow 0$.

Set up the input shares

5. For each variable w_i in the input which is sensitive (i.e. each witness variable)
 - $w_{e,i,j} \leftarrow \text{PRG}_{q_i}(\text{seed}_{e,j}, i)$ for $j \in [N]$.
 - $\phi_{e,i} \leftarrow w_i - \sum_j w_{e,i,j}$.
 - Set $\langle w_{e,i} \rangle_{q_i} \leftarrow (w_i, \{w_{e,i,j}\}_{j \in [N]})$.
6. For each variable w_i in the input which is non-sensitive set $\{w_i\} \leftarrow w_i$.
7. $\text{view}_e \leftarrow \emptyset, \text{aux}_e \leftarrow \emptyset$.

Evaluate the circuit

8. For each call to $\langle \alpha_e \rangle.\text{reveal}()$ set $\text{view}_e \leftarrow \text{view}_e \| \alpha_{e,1} \| \dots \| \alpha_{e,N}$
9. For each call to source \mathbb{S}_j with associated variables v_i of type $(q_i, *)$ execute
 - Compute $(\{\{v_i\}_{q_i}\}_\dagger, \{\{v_i\}_{q_i}\}_*, \text{aux}_{\text{cnt}_e^{\mathbb{S}_j}}^{\mathbb{S}_j}, \text{cnt}_e^{\mathbb{S}_j})$ by calling $\text{GenShares}^{\mathbb{S}}(\{\text{seed}_{e,i}^{\mathbb{S}_j}\}_\dagger, \{\text{seed}_{e,k,i}^{\mathbb{S}_j}\}_*, \text{cnt}_e^{\mathbb{S}_j})$.
 - $\text{aux}_e \leftarrow \text{aux}_e \| \text{aux}_{\text{cnt}_e^{\mathbb{S}_j}}^{\mathbb{S}_j}$.

Compute the commitments and hashes

10. Input correction terms: $(c_e^\phi, o_e^\phi) \leftarrow \text{Commit}(\{\phi_{e,i}\}_{i \in [w^*]})$.
11. Each parties seeds: $(c_{e,i}^{\text{seed}}, o_{e,i}^{\text{seed}}) \leftarrow \text{Commit}(\text{seed}_{e,i})$.
12. Correlated Randomness Correction Terms: $(c_e^{\text{aux}}, o_e^{\text{aux}}) \leftarrow \text{Commit}(\text{aux}_e)$.
13. Offline view: $h_e \leftarrow H(c_e^{\text{aux}} \| c_{e,1}^{\text{seed}} \| \dots \| c_{e,N}^{\text{seed}})$.
14. The view: $(c_e^{\text{view}}, o_e^{\text{view}}) \leftarrow \text{Commit}(\text{view}_e)$.

Compute

- $h_\phi \leftarrow H(c_1^\phi \| \dots \| c_M^\phi)$
- $h_\Gamma \leftarrow H(h_1 \| \dots \| h_M)$
- $h_{\text{view}} \leftarrow H(c_1^{\text{view}} \| \dots \| c_M^{\text{view}})$

Figure 6. The HVZK proof system for general M-Circuits

Phase 2 and Phase 3 of the MPCitH protocol

Phase 2:

On input of $E \subset [M]$ and $\{j_e\}_{e \in \bar{E}}$ from the verifier the prover computes the set of values O to open.

1. For $e \in E$ the seed to open the correlated randomness, and the material to compute h_ϕ, h_{view} , i.e. $O \leftarrow \{\text{seed}_e \| c_e^\phi \| c_e^{\text{view}}\}_{e \in E}$.
2. The number of calls to each randomness source, i.e. $O \leftarrow O \| \{\text{cnt}_i^{S_j}\}_{i \in [M], j \in [s]}$.
3. For $e \in \bar{E}$
 - All parties seeds bar the challenge one, i.e. $O \leftarrow O \| \{\text{seed}_{e,j}\}_{e \in \bar{E}, j \neq j_e}$.
 - The commitments to the missing seed, i.e. $O \leftarrow O \| \{c_{e,j_e}^{\text{seed}}\}_{e \in \bar{E}}$.
 - The opening values for the commitments to the input correction terms, i.e. $O \leftarrow O \| \{o_e^\phi\}_{e \in \bar{E}}$.
 - The opening values for the commitments to the correlated randomness correction terms, i.e. $O \leftarrow O \| \{o_e^{\text{aux}}\}_{e \in \bar{E}}$.
 - For each α_e revealed, the j_e^{th} share, i.e. $O \leftarrow O \| \{\alpha_{e,j_e}\}_{e \in \bar{E}}$.

The set of all opened values O is returned to the verifier.

Phase 3:

On input of O the verifier performs the following steps to verify the proof.

1. For $e \in E$
 - Execute the seed generation from seed_e as an honest prover would do in Phase 1.
 - Compute aux_e as an honest prover would do, and compute $h_e \leftarrow H(c_e^{\text{aux}} \| c_{e,1}^{\text{seed}} \| \dots \| c_{e,N}^{\text{seed}})$.
2. For $e \in \bar{E}$
 - For $j \neq j_e$, use $\text{seed}_{e,j}$ to compute $c_{e,j}^{\text{seed}}$ as an honest prover would, and obtain aux_e from o_e^{aux} .
 - Use the above computation with the received c_{e,j_e}^{seed} to compute h_e .
 - Use the received o_e^ϕ to compute c_e^ϕ .
 - Evaluate the circuit as an honest prover would do, using shares derived from the seeds for $j \neq j_e$, and the correction terms for the inputs and correlated randomness. For each call to $\langle \alpha_e \rangle.\text{reveal}()$ use the received α_{e,j_e} to reconstruct the non-sensitive variable $\{\alpha_e\}$. From this evaluation, compute view_e as an honest prover would do, and make sure that the output corresponds to the statement.
3. Computes h_ϕ from the received $\{c_e^\phi\}_{e \in E}$ and computed $\{c_e^\phi\}_{e \in \bar{E}}$, and compare to the received h_ϕ .
4. Compute h_Γ from the computed $\{h_e\}_{e \in [M]}$, and compare to the received h_Γ .
5. Compute h_{view} from the received $\{c_e^{\text{view}}\}_{e \in E}$ and computed $\{c_e^{\text{view}}\}_{e \in \bar{E}}$, and compare to the received h_{view} .
6. Accept the proof if all tests pass.

Figure 7. The HVZK proof system for general M-Circuits

Therefore, the overall probability that the prover can cheat successfully is,

$$\begin{aligned} \text{Adv}_{C\&C}(M, N, \tau) &= \max_{0 \leq c \leq M-\tau} \left\{ \Pr[A_c] \cdot \Pr[B_c] \right\} \\ &= \max_{0 \leq c \leq M-\tau} \left\{ \frac{\binom{M-c}{\tau}}{\binom{M}{\tau} \cdot N^{M-\tau-c}} \right\}. \end{aligned}$$

We use the notation $\text{Adv}_{C\&C}$ as our verification of the correlated randomness is performed by cut-and-choose. Later we shall look at a different method for verifying the randomness which can be applied with the correlated randomness is ‘arithmetic’ in nature.

Theorem 5.2. *Let C be an M -Circuit in the class $\mathcal{C}_{\{\mathbb{S}_1, \dots, \mathbb{S}_s\}, \emptyset}^{\{\mathbb{F}_{q_1}, \dots, \mathbb{F}_{q_f}\}}$ such that the statistical distance between Trace_C and \mathcal{S}_C is bounded by $2^{-\text{sec}}$ for all input values (with the convention that $\text{sec} = \infty$ if C is a perfectly secure representation of the desired functionality). Let H denote a collision-resistant hash function, and Commit a commitment scheme. Then the protocol in Figure 6 and Figure 7 is an HVZK argument of knowledge, with soundness error*

$$\text{Adv}_{C\&C}(M, N, \tau) + \text{Adv}_{\text{Commit}}^{\text{Binding}},$$

and with computational zero-knowledge with distinguishing advantage

$$\text{Adv}_{\text{Commit}}^{\text{Hiding}} + 2^{-\text{sec}}.$$

Proof. We show that the construction satisfies *completeness*, *honest verifier zero-knowledge*, and *(special) knowledge soundness*.

Completeness. This is straightforward and follows from the correctness of the secret sharing scheme and the description of the construction.

Honest Verifier Zero Knowledge. HVZK follows from the security of the M -Circuit, the security of the secret sharing scheme in Theorem 5.1 and by the hiding property of the commitment scheme. We construct an HVZK simulator \mathcal{S}_{ZK} which uses the simulator \mathcal{S}_{MPC} constructed in Theorem 5.1. The simulator \mathcal{S}_{ZK} works as follows:

- \mathcal{S}_{ZK} chooses random $E \subset [M]$ with size τ . Then, for each $e \in \bar{E} = [m] \setminus E$, it chooses a random $j_e \in [N]$.
- For each $e \in E$, the simulator \mathcal{S}_{ZK} acts as the honest prover, with one exception: it computes Π_e as a commitment to a 0-string.
- For each $e \in \bar{E}$, the \mathcal{S}_{ZK} chooses $\text{seed}_{e,j}$ for each $j \in I_e = [N] \setminus \{j_e\}$. Then, it generates st_e by choosing random $\Delta_{e,k}$ for each call to the correlated randomness source. In addition, it chooses random $\phi_{e,k}$ for each input wire k and compute Π_e accordingly. Then, the \mathcal{S}_{ZK} generates $view_e$ by following the instructions of \mathcal{S}_{MPC} with $\mathcal{A} = I_e$ and using $\text{seed}_{e,j}$ to generate the required randomness. For the commitments Γ_{e, \bar{j}_e} , the \mathcal{S}_{ZK} uses the 0-string as the committed message.
- For each $e \in \bar{E}$, for each $k \in o$, the \mathcal{S} sets $o_{e,k,j_e} = y_k - \sum_{j \in I_e} o_{e,j}$.
- The simulator \mathcal{S}_{ZK} computes all the hash values as an honest prover would do.
- Finally \mathcal{S}_{ZK} outputs the transcript of the protocol obtained from \mathcal{S}_{MPC} .

Since the transcript generated by \mathcal{S}_{MPC} is indistinguishable from the real transcript of the secret sharing scheme, and since the commitment scheme guarantees hiding, therefore by a hybrid argument the view generated by \mathcal{S}_{ZK} is computationally indistinguishable from the view of a real protocol instance.

(*Special Knowledge Soundness.* We assume the commitment scheme to be computationally binding and the hash function is collisions resistant. With a security loss of $\text{Adv}_{\text{Commit}}^{\text{Binding}}$ we can ‘game-hop’ to an instance where the commitment scheme is perfectly binding, thus from now on we will assume the commitment scheme is perfectly binding. We now argue that if the success probability of an adversarial prover, $\delta(x)$, is larger than soundness error $\text{Adv}_{C\&C}(M, N, \tau)$, then there exists at least one execution (out of M) where the prover has committed to a valid witness \mathbf{w} , i.e. a witness for which the evaluation of the M-Circuit C on this witness (as the sensitive inputs) gives the desired output.

Let \mathbf{G} be a binary matrix where each column corresponds to a possible challenge of the Verifier for execution (i.e. the τ executions to be revealed) and each row corresponds to one possible challenge of the Verifier for the parties (i.e. the parties indices to determine which view’s of which party are not be opened in the remaining $M - \tau$ executions). Therefore, the success probability of an adversarial prover, $\delta(x)$, is the fraction of entries equal to one in the matrix \mathbf{G} .

Let

$$\text{Adv}_{C\&C}(M, N, \tau) = \frac{\binom{M-c^*}{\tau}}{\binom{M}{\tau}} \cdot \frac{1}{N^{M-\tau-c^*}} = \frac{\binom{M-c^*}{\tau}}{\binom{M}{\tau} \cdot N^{M-\tau}} \cdot N^{c^*}$$

(i.e. c^* is the value for which the expression for $\text{Adv}_{C\&C}$ is maximized). Considering the definition of the matrix \mathbf{G} , observe that it has $\binom{M}{\tau} \cdot N^{M-\tau}$ entries. From the assumption that the success probability satisfies

$$\delta(x) > \frac{\binom{M-c^*}{\tau}}{\binom{M}{\tau} \cdot N^{M-\tau}} \cdot N^{c^*},$$

the number of one entries in the matrix \mathbf{G} is larger than $\binom{M-c^*}{\tau} \cdot N^{c^*}$.

Next, assume that in the interaction with the adversarial prover Prover^* , the prover corrupts c of the calls to the correlated randomness generations. Obviously, if any of these are revealed, then the transcript will not be accepted by the Verifier. Therefore, there can be a one entry in only in $\binom{M-c}{\tau}$ columns of the matrix \mathbf{G} . For each of these columns, there exists N^c possible challenges for the execution of the secret sharing scheme where the output of correlated randomness sources are incorrect.

Now, as there are more than $\binom{M-c^*}{\tau} \cdot N^{c^*} \geq \binom{M-c}{\tau} \cdot N^c$ with a one entry in matrix \mathbf{G} , then there must exist two accepting transcripts with the same challenge E and with different challenge $\{j_e\}_{e \in \bar{E}}$ and $\{j'_e\}_{e \in \bar{E}'}$, where $j_e \neq j'_e$ for an execution e with correct calls to the correlated randomness sources. This implies that all the views of the parties in execution e are correct as well. Therefore, the witness (the sensitive variables) used in this instance must be a valid witness \mathbf{w} .

It is therefore possible to extract the witness \mathbf{w} given two accepting transcripts $(E, \{j_e\}_{e \in \bar{E}})$, $(E', \{j'_e\}_{e \in \bar{E}'})$ when the challenge for e is different. Specifically, it is required that one of the following will hold: $e \in E \cup E' \setminus E \cap E'$ or $j'_e \neq j_e$. With the first case it is possible to extract the seeds of all parties from one transcript (where all calls to the correlated randomness sources in e are revealed) and the adjustments sent by the Prover from the second transcript (where calls to the correlated randomness sources in e are not revealed), thereby obtaining the input shares of all parties which would allow to compute w . In the other case, $j'_e \neq j_e$, one of the transcripts reveals $N - 1$ input shares whereas the other reveals the only remaining share, which would allow one to compute w by adding all shares together.

Let $\delta(x) = \text{Adv}_{C\&C}(M, N, \tau) + \epsilon$ for some $\epsilon > 0$. An extractor Ext to obtain such two transcripts can then be constructed as below.

1. Probe the matrix \mathbf{G} until the first one entry is found. Let $c = (c_1, \dots, c_M)$ be the challenge for that entry, where for each $e \in [M]$, c_e is the challenge for the j th execution.
2. For each execution e run an extractor Ext_e , who probes the matrix \mathbf{G} at random until an entry one is found for which the challenge c' is such that $c'_e \neq c_e$.
3. For each c' outputted by Ext_e , extract \mathbf{w} used in execution e using c and c' (as described before), and check that the M-Circuit on \mathbf{w} gives the desired output. If yes, output \mathbf{w} and halt.

The expected running time of the first step is $\frac{1}{\delta} < \frac{1}{\epsilon}$. For the second step, we rely on the following lemma.

Lemma 5.1. *Let $J = \{j_{e_1}, j_{e_2}, \dots\} \subseteq [M]$ be the set of indices which correspond to executions with valid witnesses. Let $|j|$ shows its size. Then there exists an $e \in J$ such that*

$$\Pr[\text{accept} | c'_e \neq c_e] \geq \frac{\epsilon}{M},$$

where accept is the event where the verifier accepts.

Proof. The proof is almost the same as in [BN20, Lemma 1], and so is omitted. \square

Note that Ext tries to extract the witness from all executions until it succeeds in extracting a valid \mathbf{w} from some execution. From the above lemma, there exists an execution e with a valid \mathbf{w} for which the probability of probing an accepting transcript that allows one to extract is larger than $\frac{\epsilon}{M}$. Therefore, the expected number of steps until the desired \mathbf{w} is extracted is bounded by $\frac{M}{\epsilon}$. The value of M depends only on the statistical security parameter but it is independent of the common input held by the Prover and the Verifier. Therefore, if the success cheating probability is larger than $\text{Adv}_{C\&C}(M, N, \tau)$, then a valid \mathbf{w} can be extracted in $O(\frac{|x|}{\epsilon})$ expected number of steps. This concludes the proof of (special) knowledge soundness. \square

5.5 Efficiency

The size of the proof depends on the proof parameters N, M and τ , which are selected to give the desired soundness error $\text{Adv}_{C\&C}(M, N, \tau)$. It also depends on the complexity of the M-Circuit C . To simplify our discussion we consider an M-Circuit over a single finite field \mathbb{F}_q , in which all correlated randomness sources require a single finite field element as the correction term generated by GenAux .

In particular if C has *online communication complexity* c_{online} (i.e. the total number of $\text{reveal}()$ operations is c_{online}), and has *offline complexity* c_{offline} (i.e. the *expected* maximum number of calls to all correlated randomness source over a valid execution³). Then the proof size will be

- Phase 1 output size $3 \cdot \ell_H$.
- Phase 2 step one: $|E| \cdot (\ell_{\text{seed}} + 2 \cdot \ell_C)$.
- Phase 2 step two: $\approx s \cdot M$.
- Phase 2 step three: $|\bar{E}| \cdot ((N-1) \cdot \ell_{\text{seed}} + \ell_C + (|\mathbf{w}^*| + c_{\text{offline}} + c_{\text{online}}) \cdot \log_2 q)$. Note, this can be reduced to $t(\log_2(N-1) \cdot \ell_{\text{seed}} + \ell_C + (|\mathbf{w}^*| + c_{\text{offline}} + c_{\text{online}}) \cdot \log_2 q)$ by generating the seeds using a tree, as explained in [BN20].

³ We say ‘expected’ as the number of bounds may not in theory be finite due to possible probabilistic behaviour, but it will be expected to be polynomially bounded if C represents an expected polynomial time function.

Where ℓ_H is the output length of the hash function H , ℓ_{seed} is the size of the seed values, and ℓ_C is the size of the commitments (all in bits). To open commitments we assume the size is equal to the size of the underlying message (which it can be for hash based commitments to high entropy bit-strings). In terms of the M-Circuit complexity this gives the overall proof size to be

$$O\left((M - \tau) \cdot (c_{\text{offline}} + c_{\text{online}}) \cdot \log_2 q\right).$$

The computational effort needed by the prover is $O(N \cdot M \cdot (c_{\text{offline}} + c_{\text{online}}))$ since the prover needs to execute the M-Circuit M times, and compute the view of each of the N players.

Independent of the circuit complexity, and hence the specific M-Circuit, we can select M , N and τ to give a desired security level $\text{Adv}_{C\&C}(M, N, \tau)$, which either minimizes communication costs, or minimizes computational cost for the prover. The following table gives the best parameters, if we allow M in the range $[30, \dots, 200]$ and N to be a power of two in the range $[2, \dots, 64]$, for different security levels.

$-\log_2 \text{Adv}_{C\&C}(M, N, \tau)$		M	N	τ
40	Best Computation	69	2	22
40	Best Communication	136	64	128
80	Best Computation	138	2	42
80	Best Communication	185	64	167
128	Best Computation	160	4	64
128	Best Communication	199	64	164

6 Using Different Correlated Randomness Sources

In the case of MPC protocols if one wants to add a new form of correlated randomness to a protocol then this equates to a more complex and costly offline phase. When using our cut-and-choose methodology for checking the correlated randomness sources in the MPCitH protocol we have already paid the cost of introducing a single source. Thus introducing new sources is essentially ‘for free’, and can indeed *reduce* complexity of this stage by requiring less data to check, as well as reducing proof complexity (both in time to produce/verify and in terms of size). Thus a new correlated randomness source should aim to reduce the online cost c_{online} , whilst not increasing c_{offline} (and the associated size of the auxiliary data needed for the resource) by a similar amount. We give two examples, one arithmetic and one non-arithmetic,

6.1 Dot-Product Computation

As an example, suppose in an M-Circuit program one is given sensitive vectors $\langle \mathbf{x} \rangle_q$ and $\langle \mathbf{y} \rangle_q$ of size k and one wishes to compute their dot-product. The naive way of doing the dot product would be to call the correlated randomness source for triple generation k times; thus receiving $\{\langle a_i \rangle_q, \langle b_i \rangle_q, \langle c_i \rangle_q\}_{i \in [k]}$, and then doing the Beaver multiplication trick k times.

$$\begin{aligned} \langle z_1 \rangle_q &= \langle c_1 \rangle_q - (x_1 - a_1) \cdot \langle b_1 \rangle_q - (y_1 - b_1) \cdot \langle a_1 \rangle_q + (x_1 - a_1) \cdot (y_1 - b_1) \\ &\dots\dots\dots \\ \langle z_k \rangle_q &= \langle c_k \rangle_q - (x_k - a_k) \cdot \langle b_k \rangle_q - (y_k - b_k) \cdot \langle a_k \rangle_q + (x_k - a_k) \cdot (y_k - b_k) \end{aligned}$$

Finally we obtain $\langle z \rangle_q = \sum_{i \in [k]} \langle z_i \rangle_q$. The k calls to the correlated randomness source, however, require k correction terms to make sure that $\langle c_i \rangle = \langle a_i \cdot b_i \rangle$ for all i . Thus these k terms need to be added to the proof. However, by introducing a different correlated randomness source tailored to this specific operation we can replace these k correction terms with a single term. To see this note that we could also write

$$\langle z \rangle_q = \sum \langle c_i \rangle_q - \sum \left((x_i - a_i) \cdot \langle b_i \rangle_q - (y_i - b_i) \cdot \langle a_i \rangle_q + (x_i - a_i) \cdot (y_i - b_i) \right).$$

Therefore, the necessary pre-processing data could be obtained by defining a new correlated source which produces values of the form

$$\langle a_1 \rangle_q, \dots, \langle a_k \rangle_q, \langle b_1 \rangle_q, \dots, \langle b_k \rangle_q, \langle c \rangle_q \text{ where } c = \sum_{i \in [k]} c_i = \sum_{i \in [k]} a_i \cdot b_i.$$

Using this source we thus need only one correction term for c , thus saving $(M - \tau) \cdot (k - 1)$ field elements of communication for the pre-processing material when using our cut-and-choose method for source correctness verification.

6.2 Matrix triples

This is a trick known in the MPC literature that can be easily applied to the setting of MPCitH. Consider $\langle \mathbf{X} \rangle_q$ and $\langle \mathbf{Y} \rangle_q$ two matrices of size $n \cdot m$ and $m \cdot l$ respectively. The naive way of computing $\langle \mathbf{Z} \rangle_q = \langle \mathbf{X} \cdot \mathbf{Y} \rangle_q$ requires $O(n \cdot m \cdot l)$ calls to the correlated randomness source for triple generation. However if one has access to a correlated randomness source for **matrix** triples $(\langle \mathbf{A} \rangle_q, \langle \mathbf{B} \rangle_q, \langle \mathbf{C} \rangle_q)$ such that $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ and \mathbf{A}, \mathbf{B} are two matrices of size $n \cdot m$ and $m \cdot l$ respectively, one can perform the matrix multiplication much more efficiently. Indeed, this source only requires $n \cdot l$ auxiliary information and the multiplication protocol is similar to the classic Beaver multiplication, thus requires to reveal one $n \cdot m$ and one $m \cdot l$ matrix.

6.3 Tiny-Tables

Interesting optimizations from the MPC world can be carried over directly to the MPCitH worlds using our abstraction. Consider for example the Tiny-Tables optimization, see [DNNR17] as extended by [KOR⁺17]. Suppose we wish, at some point in the computation, to compute a function $y = G(x)$ where $x, y \in \mathbb{F}_q$, for prime q , and x is known to be restricted to come from a small domain $\mathcal{D} \subset \mathbb{F}_q$ of size $d - 1$, with $d < q/2$. For simplicity assume $\mathcal{D} = \{0, \dots, d - 1\}$ in what follows.

We can define the correlated randomness source \mathbb{S}_G which outputs sensitive values $\langle s \rangle, \langle g_0 \rangle, \dots, \langle g_{d-1} \rangle$, subject to the constraints that s is uniformly randomly chosen from \mathbb{F}_q and that

$$g_i = G\left((s + i \pmod{q}) \pmod{d} \right).$$

Evaluation of the table on a shared value $\langle x \rangle$, whose value is guaranteed to lie in \mathcal{D} , can then be performed by opening the value $\langle h \rangle = \langle x \rangle - \langle s \rangle$, to obtain h (which we reduce into the centred interval $(-q/2, \dots, q/2)$) and then taking the result of the table look up as $\langle g_{h \pmod{d}} \rangle$.

In the case of MPCitH the size of the output of GenAux will depend on the input domain size d . Thus the Tiny-Table approach will result in a smaller proof if the table size is less than the multiplicative complexity of the function G , assuming the only alternative is to compute G via an arithmetic circuit.

7 Sacrificing

Another trick we can take from the MPC world and apply to the world of MPCitH protocols, directly from our M-Circuit definition, is that of sacrificing. At a high level one can consider the cut-and-choose component of the MPCitH protocol i.e. where the verifier selects the set E and the prover opens all the correlated randomness from the executions in E , as a method to turn a passively secure offline phase into an actively secure offline phase for the underlying MPC protocol. The method of cut-and-choose is very general, and thus applies to *any* correlated randomness source. However, some correlated randomness sources are arithmetic in nature and thus can be checked using arithmetic means. This is well known in the MPC literature, and is called sacrificing. We note a similar trick was proposed in [BN20] but not in the generality we present.

We refer to correlated randomness sources for which one can execute a method akin to sacrificing as a *Checkable Correlated Randomness Source*. Using such a check, as opposed to the generic cut-and-choose methodology from earlier, can introduce efficiencies. However, it comes at the cost of needing a five-round, as opposed to a three-round protocol.

The basic idea is to modify the correlated randomness source so that it produces an additional correlation, which is ‘sacrificed’ so as to check the correctness of the desired correlation. The correctness check makes use of a verifier defined random nonce, which can be fixed across all of the checks. To simplify our presentation we consider the case where all the variables in a randomness source are defined over the same finite field \mathbb{F}_q , where q is large; extending to smaller and different finite fields is trivial.

In terms of an M-Circuit definition, suppose we have an initial M-Circuit utilizing a desired source \mathbb{S} which produces correlated variables \mathbf{x} . However, to compile it we utilize a related source \mathbb{S}' whose output variables are of the form (\mathbf{x}, \mathbf{y}) and which has a function generating auxiliary input $\text{GenAux}^{\mathbb{S}'}$. There is then a procedure $\text{SCheck}^{\mathbb{S}}$ which takes \mathbf{x} , \mathbf{y} , the output of $\text{GenAux}^{\mathbb{S}'}$ and a challenge value t . The procedure $\text{SCheck}^{\mathbb{S}}$ which outputs a single bit b ; if $b = 0$ then the value \mathbf{x} is not from the same distribution as \mathbb{S} and if $b = 1$ then it is. The probability that the bit b is incorrect is bounded by a value $\epsilon^{\mathbb{S}'}$, with the probability being a function of the choice of the challenge value t . We say that such a source \mathbb{S}' is a Checkable Correlated Randomness Source.

7.1 The Modified MPCitH Protocol

We now modify our MPCitH protocol so as to execute the sacrificing check, as opposed to cut-and-choose, for all correlated randomness sources for which we can do so. Note, our protocol now requires five rounds of interaction. Consider an M-Circuit C in the class $\mathcal{C}_{\{\mathcal{F}\}}^{(\mathbb{S}, \emptyset)}$ where the sources \mathbb{S} are divided into two sets $\mathcal{S}_{\text{check}}$ and $\mathcal{S}_{\text{non-check}}$. For which every $\mathbb{S} \in \mathcal{S}_{\text{check}}$ there is an associated checkable source \mathbb{S}' .

Let $\mathbf{t}^{\mathbb{S}'}$ denote the random variables used in the algorithm $\text{SCheck}^{\mathbb{S}'}$. We now produce an extended M-Circuit C' which contains not only C but also the procedures $\text{SCheck}^{\mathbb{S}'}$ for all of the calls to the sources \mathbb{S} used by the original M-Circuit C . Note, that C' is an M-Circuit which takes additional input, namely the variables $\mathbf{t}^{\mathbb{S}'}$ for every $\mathbb{S} \in \mathcal{S}_{\text{check}}$. We let this new set of sources be denoted by \mathcal{S}' .

Our protocol is similar to the one before, except now the prover cannot commit to the views of the M-Circuit evaluation until it knows the verifier’s choice for $\mathbf{t}^{\mathbb{S}'}$ for every $\mathbb{S} \in \mathcal{S}_{\text{check}}$. On the other hand it must commit to the seeds generating the players secret sharings before it knows

the value of $\mathbf{t}^{\mathcal{S}'}$. This introduces an extra two rounds of communication, as we see in Figure 8 and Figure 9.

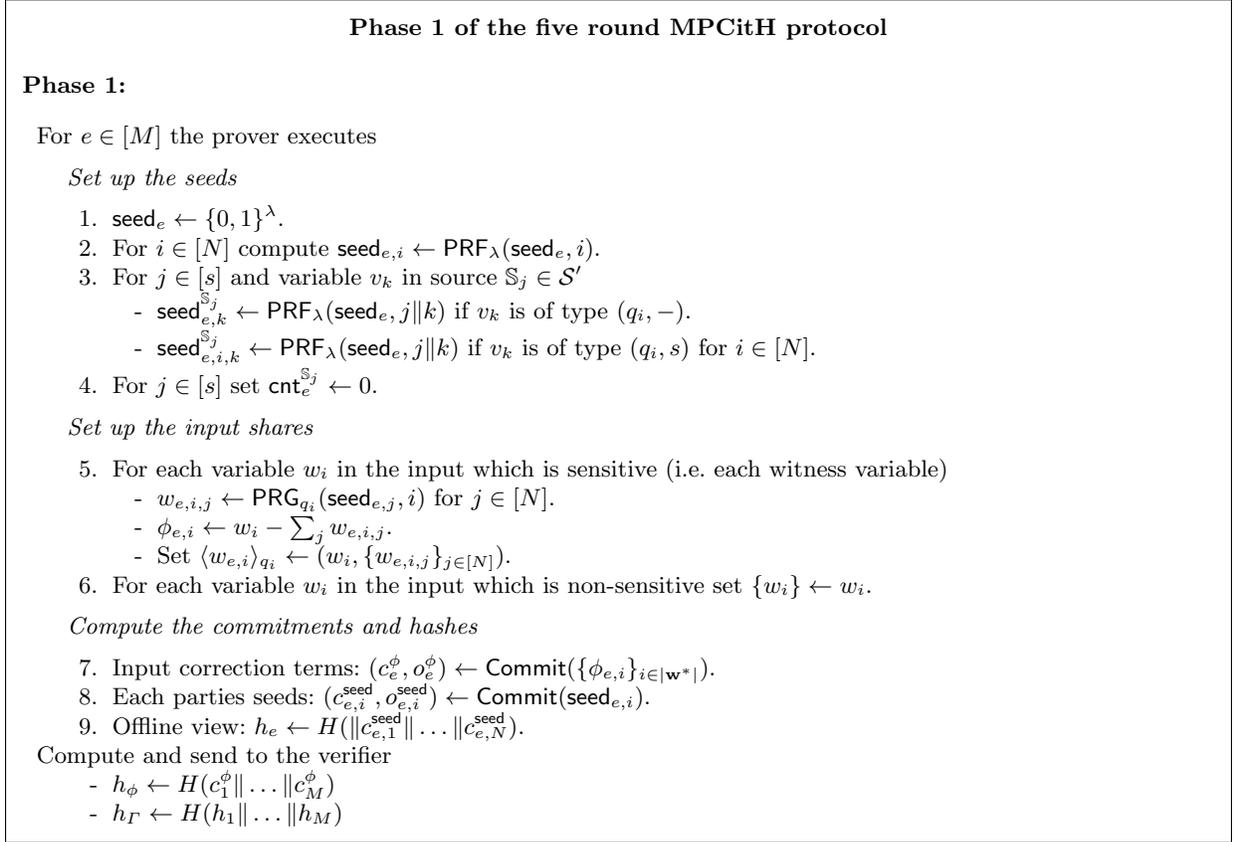


Figure 8. The five round HVZK proof system for general M-Circuits

The security of this modified protocol follows in exactly the same way as the original protocol. The only thing which is modified is the soundness probability. The value of $\Pr[A_c]$ stays the same, but we need to modify $\Pr[B_c]$ to take into account the probability that a cheating prover manages to pass the $\text{SCheck}^{\mathcal{S}'}$ checks. The adversary can either guess the challenge in the source check for a given checkable source, or he can guess which party will remain unopened; and this has to be done correctly $M - \tau - c$ times. This leads to

$$\Pr[B_c] = \left(\left(1 - \max_{\mathbb{S} \in \mathcal{S}_{\text{check}}} \epsilon^{\mathcal{S}'} \right) \frac{1}{N} + \max_{\mathbb{S} \in \mathcal{S}_{\text{check}}} \epsilon^{\mathcal{S}'} \right)^{M-\tau-c},$$

which for negligibly small values of $\epsilon^{\mathcal{S}'}$ is essentially the same as the previous value. However, when all randomness sources are checkable we can set $\tau = 0$ as explained above, and our soundness becomes

$$\text{Adv}_{\text{Sacrifice}}(M, N) = \left(\left(1 - \max_{\mathbb{S} \in \mathcal{S}_{\text{check}}} \epsilon^{\mathcal{S}'} \right) \frac{1}{N} + \max_{\mathbb{S} \in \mathcal{S}_{\text{check}}} \epsilon^{\mathcal{S}'} \right)^M$$

We note that to determine how many calls we make to the random sources, we need to know the value of all the variables in the M-circuit. Therefore, the output of all sources (checkable and non-checkable) must be known before receiving the cut-and-choose challenge. An attentive reader

Phase 2, 3, and 4 of the five round MPCitH protocol

Phase 2:

On input of $\mathbf{t}^{\mathbb{S}'}$ from the verifier for every checkable source $\mathbb{S} \in \mathcal{S}_{\text{check}}$

For $e \in [M]$ the prover executes

1. $\text{view}_e \leftarrow \emptyset, \text{aux}_e \leftarrow \emptyset$.

Evaluate the circuit C'

2. For each call to $\langle \alpha_e \rangle.\text{reveal}()$ set $\text{view}_e \leftarrow \text{view}_e \parallel \alpha_{e,1} \parallel \dots \parallel \alpha_{e,N}$
3. For each call to source \mathbb{S}'_j such that $\mathbb{S}_j \in \mathcal{S}_{\text{check}}$ and with associated variables v_i of type $(q_i, *)$ execute
 - Compute $(\{\{v_i\}_{q_i}\}_{\dagger}, \{\{v_i\}_{q_i}\}_{*}, \text{aux}_{\text{cnt}_e^{\mathbb{S}'_j}}^{\mathbb{S}'_j}, \text{cnt}_e^{\mathbb{S}'_j})$ by calling $\text{GenShares}^{\mathbb{S}'_j}(\{\text{seed}_{e,i}^{\mathbb{S}'_j}\}_{\dagger}, \{\text{seed}_{e,k,i}^{\mathbb{S}'_j}\}_{*}, \text{cnt}_e^{\mathbb{S}'_j})$.
 - $\text{aux}_e \leftarrow \text{aux}_e \parallel \text{aux}_{\text{cnt}_e^{\mathbb{S}'_j}}^{\mathbb{S}'_j}$.
4. For each call to source $\mathbb{S}_j \in \mathcal{S}_{\text{non-check}}$ and with associated variables v_i of type $(q_i, *)$ execute
 - Compute $(\{\{v_i\}_{q_i}\}_{\dagger}, \{\{v_i\}_{q_i}\}_{*}, \text{aux}_{\text{cnt}_e^{\mathbb{S}_j}}^{\mathbb{S}_j}, \text{cnt}_e^{\mathbb{S}_j})$ by calling $\text{GenShares}^{\mathbb{S}_j}(\{\text{seed}_{e,i}^{\mathbb{S}_j}\}_{\dagger}, \{\text{seed}_{e,k,i}^{\mathbb{S}_j}\}_{*}, \text{cnt}_e^{\mathbb{S}_j})$.
 - $\text{aux}_e \leftarrow \text{aux}_e \parallel \text{aux}_{\text{cnt}_e^{\mathbb{S}_j}}^{\mathbb{S}_j}$.

Commit to the sources and the views

5. Correlated Randomness Correction Terms: $(c_e^{\text{aux}}, o_e^{\text{aux}}) \leftarrow \text{Commit}(\text{aux}_e)$.
6. $(c_e^{\text{view}}, o_e^{\text{view}}) \leftarrow \text{Commit}(\text{view}_e)$.

Compute and send to the verifier

- $h_{\text{view}} \leftarrow H((c_e^{\text{aux}} \parallel c_1^{\text{view}} \parallel \dots \parallel c_M^{\text{view}}))$

Phase 3:

On input of $E \subset [M]$ and $\{j_e\}_{e \in \bar{E}}$ from the verifier this proceeds exactly as in the Phase 2 of the original protocol from Figure 7.

Phase 4:

On input of O the verifier performs proceeds exactly as in the Phase 3 of the original protocol from Figure 7; except that h_e is computed without c_e^{aux} , and c_e^{aux} is used to compute h_{view} . The verifier will also reject the proof however if any output from a $\text{SCheck}^{\mathbb{S}'}$ operation returns zero whilst evaluating the M-Circuits.

Figure 9. The five round HVZK proof system for general M-Circuits

may thus point out that if one has to pay the soundness cost of cut-and-choose, then there is no practical reason for adding sacrificing on top of it, as all correlated randomness sources that can be checked by sacrificing can also be checked by cut-and-choose. This is indeed the case, and in the case of an M-Circuit $C \in \mathcal{C}_{\mathcal{F}}^{(\mathcal{S}, \emptyset)}$, one should, in practice, use the sacrificing technique only if $\forall \mathcal{S} \in \mathcal{S}, \mathcal{S} \in \mathcal{S}_{\text{check}}$, and completely ignore the cut-and-choose part of the protocol (set $\tau = 0$).

However the usefulness of our 5-round protocol will be clear once we describe gadgets. Indeed, gadgets will be treated in a similar way as checkable correlated randomness sources, with the exception that, unlike correlated randomness sources, the cut-and-choose technique can not be used to check the correctness of their execution.

Example Checkable Correlated Randomness Sources: Note, the ‘program’ for the check $\text{SCheck}^{\mathcal{S}}$ can be expressed as an M-Circuit in the class $\mathcal{C}_{\mathbb{F}_q}^{(\emptyset, \emptyset)}$. Thus property of being a checkable correlated randomness source is a function not only of the procedure $\text{SCheck}^{\mathcal{S}}$ existing, but also of the definition of the function $\text{GenAux}^{\mathcal{S}'}$ as we now illustrate.

Triple. Recall the source **Triple** produces tuples $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ such that $c = a \cdot b$. Our ‘extended’ source **Triple’** produces values $(\langle a \rangle, \langle b \rangle, \langle c \rangle, \langle d \rangle, \langle e \rangle)$ where it is ‘claimed’ that $c = a \cdot b$ and $e = b \cdot d$. This validity can be verified by the following algorithm which takes as input a public value $t \in \mathbb{F}_q$ (which can be the same for every output of **Triple’**).

1. $\langle \rho \rangle \leftarrow t \cdot \langle a \rangle - \langle d \rangle$.
2. $\rho \leftarrow \langle \rho \rangle.\text{reveal}()$.
3. $\langle r \rangle \leftarrow t \cdot \langle c \rangle - \langle e \rangle - \rho \cdot \langle b \rangle$.
4. $r \leftarrow \langle r \rangle.\text{reveal}()$.
5. Reject if $r \neq 0$.

Note, this algorithm is an M-Circuit in the class $\mathcal{C}_{\mathbb{F}_q}^{(\emptyset, \emptyset)}$. Also note that the algorithm reveals no information about the values $\langle a \rangle, \langle b \rangle$ and $\langle c \rangle$. Also note, that for a valid tuple we have $r = t \cdot c - e - \rho \cdot b = t \cdot a \cdot b - b \cdot d - (t \cdot a - d) \cdot b = 0$, and note that if $c \neq a \cdot b$ and $e \neq b \cdot d$ then we have $r = t \cdot (c - a \cdot b) + e + b \cdot d$ which will equal zero with probability $\epsilon^{\text{Triple}'} = 1/q$, when t is chosen independently of the output of **Triple’**.

However, whilst this verifies that the $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ variables output by **Triple’** satisfy the desired multiplicative relationship, it does not on its own demonstrate that the distribution of $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ is correct; namely that $\langle a \rangle$ and $\langle b \rangle$ are chosen uniformly at random. To ensure this we need to examine how $\text{GenAux}^{\text{Triple}'}$ is defined. Mirroring our two previous instantiations of $\text{GenAux}^{\text{Triple}}$ we have

1. $\text{GenAux}^{\text{Triple}'}(a, b, c, d, e) = (0, 0, a \cdot b - c, b \cdot d - e)$.
2. $\text{GenAux}^{\text{Triple}'}(a, b, c, d, e) = (x - a, y - b, z - c, u - d, w - e)$ where x, y, u are deterministically selected from \mathbb{F}_p by $\text{GenAux}^{\text{Triple}'}$ using a PRG with the seed $H(a, b, c, d, e)$, for some hash function H , $z = x \cdot y$ and $w = y \cdot u$.

The first case produces the correct distribution irrespective of what the prover computes, whereas the second case does not. In the second case a cheating prover can deviate from the protocol and make $\langle a \rangle$ follow any distribution they desire.

Square. Recall this works in roughly the same way, the source **Square** produces pairs of values $(\langle a \rangle, \langle b \rangle)$ such that $b = a \cdot a$. Our ‘extended’ source **Square’** produces values $(\langle a \rangle, \langle b \rangle, \langle f \rangle, \langle h \rangle)$ where it is ‘claimed’ that $b = a \cdot a$ and $h = f \cdot f$. This validity can be verified by the following algorithm which takes as input a public value $t \in \mathbb{F}_q$.

1. $\langle \rho \rangle \leftarrow t \cdot \langle a \rangle - \langle f \rangle$.
2. $\rho \leftarrow \langle \rho \rangle.\text{reveal}()$.
3. $\langle r \rangle \leftarrow t^2 \cdot \langle b \rangle - \langle h \rangle - \rho \cdot (t \cdot \langle a \rangle + \langle f \rangle)$.
4. $r \leftarrow \langle r \rangle.\text{reveal}()$.
5. Reject if $r \neq 0$.

Again the same analysis shows that this checks correctness and an adversary can make the check pass with probability $\epsilon^{\text{Square}'} = 1/q$.

Bit. As remarked earlier this is the more interesting correlated randomness source in applications, as it enables far more efficient M-Circuit representations of functions. The source **Bit** produces a value $\langle b \rangle$ such that b is guaranteed to lie in $\{0, 1\}$. However, whilst in an MPC protocol there is a sacrificing methodology for Bits, this does not translate over to the MPCitH paradigm as one needs a way of verifying the bits are uniformly selected. Thus checking the source **Bit** seems to require cut-and-choose.

8 Executable Gadgets

Up until now we have considered for our MPCitH protocols only M-Circuits from classes of the form $\mathcal{C}_{\mathcal{F}}^{(\mathcal{S}, \emptyset)}$, i.e. M-Circuits with no gadgets. A gadget captures an essential non-linear subroutine within an M-Circuit. By abstracting it away, we simplify the composition of special-purpose protocols for such subroutines within a more generic M-Circuit. Whilst M-Circuits can describe arbitrary gadgets, only special gadgets, which we call *executable gadgets* are able to be supported by the MPCitH protocol.

We proceed to the formal definition of an executable gadget, which we define over a single finite field \mathbb{F}_q of large characteristic for ease of exposition, and then we present two examples of executable gadgets for MPCitH protocols.

Definition 8.1 (Executable Gadget). *An Executable Gadget \mathbb{G} is an object defined by*

- I. *A function G with (possibly zero) inputs and (at least one) output in \mathcal{F} .*
- II. *A $\text{GenAux}^{\mathbb{G}}$ function that fixes the auxiliary information needed to correct a uniformly random \mathbf{y} to be equal to $G(\mathbf{x})$, i.e. $\text{GenAux}^{\mathbb{G}}(\mathbf{x}, \mathbf{y}) = G(\mathbf{x}) - \mathbf{y}$.*
- III. *A $\text{GCheck}^{\mathbb{G}}$ M-Circuit in the class $\mathcal{C}_{\mathcal{F}}^{(\mathcal{S}, \emptyset)}$ for a set of randomness sources \mathcal{S} , the function which takes as input \mathbf{x} , \mathbf{y} , the output of $\text{GenAux}^{\mathbb{G}}(\mathbf{x}, \mathbf{y})$ and a challenge value $t \in \mathcal{F}$. The procedure $\text{GCheck}^{\mathbb{G}}$ which outputs a single bit b ; if $b = 0$ then the values are inconsistent, i.e. the purported value of \mathbf{aux} is not correct, and if $b = 1$ then it is correct. The probability that the bit b is incorrect is bounded by a value $\epsilon^{\mathbb{G}}$, with the probability being a function of the choice of the challenge value t .*

Thus an executable gadget is very similar to the checkable randomness sources from the previous section. To process the gadget within the MPCitH protocol we thus proceed just as we did for checkable randomness sources; the initial M-Circuit C is extended to an augmented circuit C'

which includes all the necessary $\text{GCheck}^{\mathbb{G}}$ operations. As $\text{GCheck}^{\mathbb{G}}$ itself potentially requires access to correlated randomness sources this might require the addition of additional correlated randomness source. Then the five round protocol is executed, so that the augmented circuit C' can be created (as it depends on the verifier's selection of the challenges t in $\text{GCheck}^{\mathbb{G}}$). The modification to the soundness error is the same as that introduced for checkable randomness sources.

The BitDecomp Gadget: The executable gadget **BitDecomp** for a given sensitive value $\langle x \rangle_q$ produces the $\lceil \log_2(q) \rceil$ sensitive values $\langle b_i \rangle_q$ such that $b_i \in \{0, 1\}$ and $x = \sum_{i=0}^{\lceil \log_2(q) \rceil - 1} b_i \cdot 2^i$. A simple example is the bit decomposition operation $(\langle b_0 \rangle_q, \dots, \langle b_{\lceil \log_2(q) \rceil - 1} \rangle_q) \leftarrow \text{BitDecomp}(\langle x \rangle_q)$, where $\forall i, b_i \in \{0, 1\}$ and $\sum_i b_i 2^i = x$. See Figure 10 for a formal specification, note this checking procedure requires no random input from the verifier; this is because there is implicitly random input needed to check the (checkable) correlated randomness source **Square** which it requires.

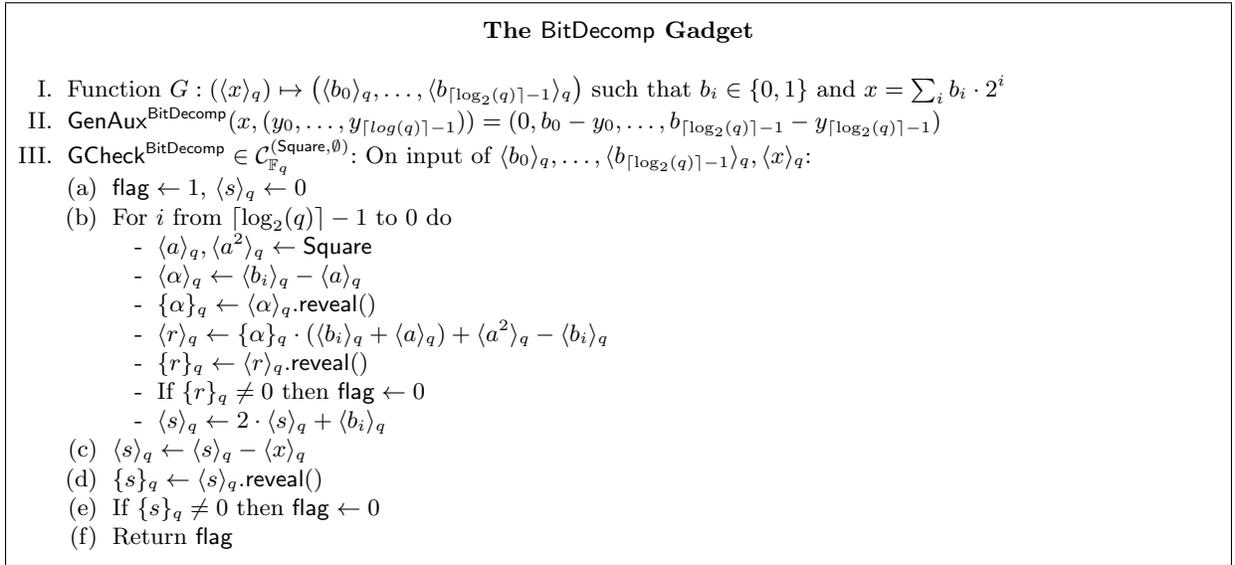


Figure 10. The BitDecomp Gadget

In the above instantiation of **BitDecomp** we assumed that our randomness source **Square** was already checked by either cut-and-choose or sacrificing. However, we can obtain a further efficiency if we merge the checking of the output of **Square** with the checking of this bits produced in the gadget. To present this we give utilize a correlated randomness source **USquare**, which represents an unchecked square tuple. Namely, we check the output is correct neither by the sacrificing style check or via cut-and-choose. This allows us to present an more efficient check of the **BitDecomp Gadget** in Figure 11, where now we require the verifier to provide a random challenge $t \in \mathbb{F}_q$. At first sight it seems to involve the same number of reveals operations, but we actually save operations as we no longer need reveals to check the output of **Square**.

The RNSDecomp Gadget: If we extend our definitions to rings \mathbb{Z}_q with $q = \prod_{i=1}^k p_i$ and p_i primes, an interesting technique which has been widely used in cryptography is to make use of the Chinese Remainder Theorem. In MPCitH it is very easy for the prover to inject the residues of a sensitive variable such that the M-Circuit can operate on those residues. Since CRT reconstruction

The Optimized BitDecomp Gadget

- I. Function $G : (\langle x \rangle_q) \mapsto (\langle b_0 \rangle_q, \dots, \langle b_{\lceil \log_2(q) \rceil - 1} \rangle_q)$ such that $b_i \in \{0, 1\}$ and $x = \sum_i b_i \cdot 2^i$
- II. $\text{GenAux}^{\text{BitDecomp}}(x, (y_0, \dots, y_{\lceil \log_2(q) \rceil - 1})) = (0, b_0 - y_0, \dots, b_{\lceil \log_2(q) \rceil - 1} - y_{\lceil \log_2(q) \rceil - 1})$
- III. $\text{GCheck}^{\text{BitDecomp}} \in \mathcal{C}_{\mathbb{F}_q}^{\text{Square}, \emptyset}$: On input of $\langle b_0 \rangle_q, \dots, \langle b_{\lceil \log_2(q) \rceil - 1} \rangle_q, \langle x \rangle_q$ and a value t from the verifier:
 - (a) $\text{flag} \leftarrow 1, \langle s \rangle_q \leftarrow 0$
 - (b) For i from $\lceil \log_2(q) \rceil - 1$ to 0 do
 - $\langle a \rangle_q, \langle a^2 \rangle_q \leftarrow \text{USquare}$
 - $\langle \rho \rangle_q \leftarrow \langle b_i \rangle_q - t \cdot \langle a \rangle_q$
 - $\{\rho\}_q \leftarrow \langle \rho \rangle_q \cdot \text{reveal}()$
 - $\langle r \rangle_q \leftarrow \langle b_i \rangle_q - \{\rho\}_q \cdot (\langle b_i \rangle_q + t \cdot \langle a \rangle_q) - t^2 \cdot \langle a^2 \rangle_q$.
 - $\{r\}_q \leftarrow \langle r \rangle_q \cdot \text{reveal}()$
 - If $\{r\}_q \neq 0$ then $\text{flag} \leftarrow 0$
 - $\langle s \rangle_q \leftarrow 2 \cdot \langle s \rangle_q + \langle b_i \rangle_q$
 - (c) $\langle s \rangle_q \leftarrow \langle s \rangle_q - \langle x \rangle_q$
 - (d) $\{s\}_q \leftarrow \langle s \rangle_q \cdot \text{reveal}()$
 - (e) If $\{s\}_q \neq 0$ then $\text{flag} \leftarrow 0$
 - (f) Return flag

Figure 11. The Optimized BitDecomp Gadget

is a linear operation, it is also trivial to design a GCheck M-Circuit, as it suffice to apply the linear CRT reconstruction algorithm to the residues, and compare the result with the original value. An application of such a technique would then be to use the Tiny-Tables optimization described previously, but for functions with domain \mathbb{Z}_q that can be computed residue-wise. By following the blueprint of [BMR16], one would then create a table of the desired function for all the residues, thus going from a prohibitive size q table to k tables of total size $\sum p_i$. (e.g. exponentiation of a sensitive variable by a non-sensitive variable)

The RNSDecomp get

- I. Function $G : (\langle x \rangle_q) \mapsto (\langle x_1 \rangle_{p_1}, \dots, \langle x_k \rangle_{p_k})$ such that $x_i \in \mathbb{F}_{p_i}$ and $x = \text{CRT}([x_1, \dots, x_k], [p_1, \dots, p_k])$
- II. $\text{GenAux}^{\text{RNSDecomp}}(x, (y_1, \dots, y_k)) = (0, x_1 - y_1, \dots, x_k - y_k)$
- III. $\text{GCheck}^{\text{RNSDecomp}} \in \mathcal{C}_{\mathbb{Z}_q, \mathbb{F}_{p_1}, \dots, \mathbb{F}_{p_k}}^{\emptyset, \emptyset}$: On input of $\langle x_1 \rangle_{p_1}, \dots, \langle x_k \rangle_{p_k}, \langle x \rangle_q$:
 - (a) $\text{flag} \leftarrow 1$
 - (b) $\langle s \rangle_q \leftarrow \text{CRT}([\langle x_1 \rangle_{p_1}, \dots, \langle x_k \rangle_{p_k}], [p_1, \dots, p_k])$ (Local operation)
 - (c) $\langle s \rangle_q \leftarrow \langle s \rangle_q - \langle x \rangle_q$
 - (d) $\{s\}_q \leftarrow \langle s \rangle_q \cdot \text{reveal}()$
 - (e) If $\{s\}_q \neq 0$ then $\text{flag} \leftarrow 0$
 - (f) Return flag

Figure 12. Residue Number System Decomposition

Acknowledgments

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. HR001120C0085 and FA8750-19-C-0502, by the FWO under an Odysseus project GOH9718N, and by CyberSecurity Research Flanders with reference number VR20192203.

References

- ACK⁺20. Abdelrahman Aly, Kelong Cong, Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Oliver Scherer, Peter Scholl, Nigel P. Smart, Titouan Tanguy, and Tim Wood. SCALE and MAMBA v1.9: Documentation, 2020.
- AHIV17. Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkatasubramanian. Ligerio: Lightweight sublinear arguments without a trusted setup. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 2087–2104, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.
- AOR⁺19. Abdelrahman Aly, Emmanuela Orsini, Dragos Rotaru, Nigel P. Smart, and Tim Wood. Zaphod: Efficiently combining LSSS and garbled circuits in SCALE. In Michael Brenner, Tancrede Lepoint, and Kurt Rohloff, editors, *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography, WAHC@CCS 2019, London, UK, November 11-15, 2019*, pages 33–44. ACM, 2019.
- Bea92. Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology – CRYPTO’91*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432, Santa Barbara, CA, USA, August 11–15, 1992. Springer, Heidelberg, Germany.
- BFH⁺20. Rishabh Bhadauria, Zhiyong Fang, Carmit Hazay, Muthuramakrishnan Venkatasubramanian, Tiancheng Xie, and Yupeng Zhang. Ligerio++: A new optimized sublinear IOP. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 20: 27th Conference on Computer and Communications Security*, pages 2025–2038, Virtual Event, USA, November 9–13, 2020. ACM Press.
- BGW88. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 1–10, Chicago, IL, USA, May 2–4, 1988. ACM Press.
- BL73. David Elliott Bell and Leonard J. LaPadula. Secure computer systems: Mathematical foundations. MITRE Corporation Technical Report 2547, 1973.
- 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 2016: 23rd Conference on Computer and Communications Security*, pages 565–577, Vienna, Austria, October 24–28, 2016. ACM Press.
- BN20. Carsten Baum and Ariel Nof. Concretely-efficient zero-knowledge arguments for arithmetic circuits and their application to lattice-based cryptography. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020: 23rd International Conference on Theory and Practice of Public Key Cryptography, Part I*, volume 12110 of *Lecture Notes in Computer Science*, pages 495–526, Edinburgh, UK, May 4–7, 2020. Springer, Heidelberg, Germany.
- CCD88. David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 11–19, Chicago, IL, USA, May 2–4, 1988. ACM Press.
- Cd10. Octavian Catrina and Sebastiaan de Hoogh. Improved primitives for secure multiparty integer computation. In Juan A. Garay and Roberto De Prisco, editors, *SCN 10: 7th International Conference on Security in Communication Networks*, volume 6280 of *Lecture Notes in Computer Science*, pages 182–199, Amalfi, Italy, September 13–15, 2010. Springer, Heidelberg, Germany.
- 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, *Advances in Cryptology – CRYPTO 2018, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 769–798, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.
- CDG⁺17. Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Reicherberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 1825–1842, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.
- CGH⁺18. Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 34–64, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.
- CS10. Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In Radu Sion, editor, *FC 2010: 14th International Conference on Financial Cryptography and Data Security*, volume

- 6052 of *Lecture Notes in Computer Science*, pages 35–50, Tenerife, Canary Islands, Spain, January 25–28, 2010. Springer, Heidelberg, Germany.
- DFK⁺06. Ivan Damgård, Matthias Fitz, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *TCC 2006: 3rd Theory of Cryptography Conference*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304, New York, NY, USA, March 4–7, 2006. Springer, Heidelberg, Germany.
- DGKN09. Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009: 12th International Conference on Theory and Practice of Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179, Irvine, CA, USA, March 18–20, 2009. Springer, Heidelberg, Germany.
- 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 Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013: 18th European Symposium on Research in Computer Security*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18, Egham, UK, September 9–13, 2013. Springer, Heidelberg, Germany.
- DNNR17. Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. The TinyTable protocol for 2-party secure computation, or: Gate-scrambling revisited. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 167–187, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany.
- DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.
- FS87. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO’86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194, Santa Barbara, CA, USA, August 1987. Springer, Heidelberg, Germany.
- GMO16. Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster zero-knowledge for Boolean circuits. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016: 25th USENIX Security Symposium*, pages 1069–1083, Austin, TX, USA, August 10–12, 2016. USENIX Association.
- IKOS07. Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th Annual ACM Symposium on Theory of Computing*, pages 21–30, San Diego, CA, USA, June 11–13, 2007. ACM Press.
- Kel20. Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 20: 27th Conference on Computer and Communications Security*, pages 1575–1590, Virtual Event, USA, November 9–13, 2020. ACM Press.
- KKW18. Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and Xiaofeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 525–537, Toronto, ON, Canada, October 15–19, 2018. ACM Press.
- KOR⁺17. Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Eduardo Soria-Vazquez, and Srinivas Vivek. Faster secure multi-party computation of AES and DES using lookup tables. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17: 15th International Conference on Applied Cryptography and Network Security*, volume 10355 of *Lecture Notes in Computer Science*, pages 229–249, Kanazawa, Japan, July 10–12, 2017. Springer, Heidelberg, Germany.
- KOS16. Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 830–842, Vienna, Austria, October 24–28, 2016. ACM Press.
- RW19. Dragos Rotaru and Tim Wood. MArBled circuits: Mixing arithmetic and Boolean circuits with active security. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *Progress in Cryptology - INDOCRYPT 2019: 20th International Conference in Cryptology in India*, volume 11898 of *Lecture Notes in Computer Science*, pages 227–249, Hyderabad, India, December 15–18, 2019. Springer, Heidelberg, Germany.