Computer-aided proofs for multiparty computation with active security

Helene Haagh* Aleksandr Karbyshev* Sabine Oechsner* Bas Spitters* Pierre-Yves Strub†
*Aarhus University, DK
†École Polytechnique, F

***Abstract*—Secure multi-party computation (MPC) is a general cryptographic technique that allows distrusting parties to compute a function of their individual inputs, while only revealing the output of the function. It has found applications in areas such as auctioning, email filtering, and secure teleconference.**

Given their importance, it is crucial that the protocols are specified and implemented correctly. In the programming language community, it has become good practice to use computer proof assistants to verify correctness proofs. In the field of cryptography, EasyCrypt is the state of the art proof assistant. It provides an embedded language for probabilistic programming, together with a specialized logic, embedded into an ambient general purpose higher-order logic. It allows us to conveniently express cryptographic properties. EasyCrypt has been used successfully on many applications, including public-key encryption, signatures, garbled circuits and differential privacy. Here we show for the first time that it can also be used to prove security of MPC against a *malicious* adversary.

We formalize additive and replicated secret sharing schemes and apply them to Maurer's MPC protocol for secure addition and multiplication. Our method extends to general polynomial functions. We follow the insights from EasyCrypt that security proofs can often be reduced to proofs about program equivalence, a topic that is well understood in the verification of programming languages. In particular, we show that for a class of MPC protocols in the passive case the non-interference-based (NI) definition is equivalent to a standard simulation-based security definition. For the active case, we provide a new non-interference based alternative to the usual simulation-based cryptographic definition that is tailored specifically to our protocol.

I. Introduction

The study of multiparty computation started in the 1980s with the work of Yao [30] and Goldreich et al. [18]. It has since grown increasingly important and is starting to be used in real-life applications such as auctioning [17], email filtering, secure teleconference [23].

A widely used technique for constructing MPC protocols is secret sharing [29, 11], a cryptographic primitive that distributes a secret among several parties by providing each party with a share of the secret. The secret can be reconstructed by combining the shares belonging to a qualified subset of the parties (i.e. parties that are allowed to learn the secret), while other subsets of the parties will have no information on the secret (even when combining their shares).

EasyCrypt [6, 5] has been used to verify cryptographic primitives, and more recently to verify protocols by using it's built-in probabilistic While-language. While cryptography papers usually provide a presentation of the algorithms in pseudocode, EasyCrypt code of such a protocol is usually not much longer, but has the benefit of being completely precise. Moreover, a framework is being developed to tie EasyCrypt into a fully verified tool chain to generate verified low level code from the protocol definition [1] and thus obtain high assurance cryptography.

A clear motivation for formal verification is given by Bellare and Rogaway [8]: 'In our opinion, many proofs in cryptography have become essentially unverifiable. Our field may be approaching a crisis of rigor.' An example of the usefulness of formal verification is provided by the vulnerabilities in the Dual EC random bit generator, where the correctness proof was flawed [19]. Any attempt to formalize this argument would have spotted the gap in the proof. In particular, an attacker that chooses the constants used in Dual EC could potentially predict outputs and this way introduce a backdoor into protocols using Dual EC such as TLS [16]. The *feasibility* of verification is, for instance, demonstrated by the subsequent verification of improved protocols for elliptic curve cryptography [31]. Finally, formal verification is required to obtain the highest assurance level (EAL7) in Common Criteria.

A. Our Contribution

- We provide security definitions and proofs for the MPC protocol by Maurer [25]. This is the first formalized proof for more than two parties and the first formalized proof of a protocol that is *actively* secure.
- We have formalized these proofs in EasyCrypt, a tool that has been used for cryptographic primitives, but only recently also for protocols. A precise description of what we have formalized is presented in section V-D.

We split the protocol into three phases: input, computation and output, where the computation phase can potentially consist of an arbitrary combination of additions and multiplications. This standard approach allows us to treat arithmetic circuits. Mathematically, one can see arithmetic circuits as a way to represent multi-variate polynomials on the ring[1] $\mathbb{Z}_m$, and therefore, to represent any function over $\mathbb{Z}_m$.

We first discuss simulation-based security definitions for passive and active security of MPC protocols. The use of simulation-based notions is the standard approach in cryptography for defining security of MPC protocols We then proceed with new non-interference-based definitions that are tailored

[1]In the cryptographic literature, most MPC protocols rely on computation over a field. In the case of our protocol, however, a ring is sufficient.

specifically to the class of protocols we consider, and relate them to the simulation-based ones. In particular, we prove that they imply the standard simulation-based cryptographic definitions. In the passive case, non-interference is equivalent to the existence of a simulator. Non-interference (NI) is especially suitable for the computer-aided proofs presented in EasyCrypt, where the probabilistic relational Hoare logic presents a solid foundation for proving non-interference based statements. A simulation-based proof would proceed by considering an equivalence between a program and a simulator, which are structurally different, whereas NI considers two runs of the *same* program. In the active case, this difference is even bigger because in the security definition the simulator does not obtain the protocol output in advance. The feasibility of using non-interference for cryptography is known and was e.g. studied by Backes and Pfitzmann [3]. In the case of EasyCrypt, the benefits were emphasized, in a different context, in a work on masking schemes by Barthe et al. [4]. NI is a compositional property, which allows us to build modular proofs.

An important motivation for our work is provided by the EasyCrypt formalization of Boolean garbled circuits [1]. It provides high assurance crypto, a completely verified tool chain starting from a readable EasyCrypt protocol to verified low level code. We hope to profit from the same technology in the future. Garbled circuits provide a framework for secure 2-party computation, a technique complementary to the techniques we use here.

B. Outline

Section III contains background on secure multi-party computation. Section IV contrasts non-interference based definitions with simulation based ones. Section V discusses our modelling of Maurer's description of active security for the addition protocol. Section VI discusses related work and Section VII concludes.

The sources are available at http://users-cs.au.dk/spitters/MPCEC/. The implementation compiles with the development version of EasyCrypt available from https://github.com/EasyCrypt/easycrypt.

II. Preliminaries

Let $[n]$ denote the set $\{1, \ldots, n\}$. We will use $\mathbf{x}$ to denote a vector $(x_1, \ldots, x_n)$ and $\mathbf{x}_i$ for its ith projection. Conveniently, almost all vectors in this paper have the same length. Let $x \leftarrow \mathcal{D}$ denote the sampling of an element x according to some distribution $\mathcal{D}$, and let $x \leftarrow_\$ S$ denote that x is sampled from the uniform distribution over the finite set S. We fix an integer m and consider the ring $\mathbb{Z}_m$.

We now want to compare the output distributions of two executions of probabilistic algorithms.

Definition 1 (Perfect indistinguishability)**.** *Let $U(x)$ and $V(x)$ be the output distribution of probabilistic algorithm U and V, respectively, on input x. Then U and V are perfectly indistinguishable (denoted $U \sim^p V$) if for all inputs x, $U(x) = V(x)$.*

A. Secret Sharing

Here we recall the definitions of additive, replicated, and verifiable secret sharing schemes from [25]. The definitions build on top of each other: The verifiable secret sharing scheme combines replicated secret sharing with additional communication, and replicated secret sharing uses additive secret sharing internally. Looking ahead, we want to state the definition of our replicated and verifiable secret sharing scheme for the special secrecy structure[2] with privacy against a single party[3]. I.e. let $\mathcal{P} = \{P_1, \ldots, P_n\}$ be the set of parties, then for each $i \in [n]$ $\{P_i\}$ is in the secrecy structure Σ. The replicated secret sharing scheme with this access structure requires the underlying additive secret sharing scheme to provide privacy against any set of $n-1$ colluding parties. Throughout this work, we will refer to the party that creates a secret sharings as the *dealer* (of the corresponding secret).

a) Additive Secret Sharing.: The additive secret sharing scheme for n parties consist of a pair of algorithms:

$$\mathsf{ASS} = (\textsc{Share}, \textsc{Reconstruct}),$$

which are defined as follows:

Sharing: The $\mathsf{ASS}.\textsc{Share}$ algorithm on input a secret $s \in \mathbb{Z}_p$, samples $n-1$ values $a_1, \ldots, a_{n-1}$ and computes $a_n = s - \sum_{i=1}^{n-1} a_i$. Then it outputs the shares $(a_1, \ldots, a_n)$.

Reconstruct: The $\mathsf{ASS}.\textsc{Reconstruct}$ algorithm on input a sharing $(a_1, \ldots, a_n)$, computes and outputs $s = \sum_{i=1}^{n} a_i$.

b) Replicated Secret Sharing.: The replicated secret sharing scheme for n parties consist of a pair of algorithms:

$$\mathsf{RepSS} = (\textsc{Share}, \textsc{Reconstruct}),$$

which are defined as follows:

Sharing: The $\mathsf{RepSS}.\textsc{Share}$ algorithm on input a secret $s \in \mathbb{Z}_p$, runs the additive sharing algorithm $(a_1, \ldots, a_n) \leftarrow \mathsf{ASS}.\textsc{Share}(s)$. Then it constructs for all $i \in [n]$

$$r_i = (a_1, \ldots, a_{i-1}, \bot, a_{i+1}, \ldots, a_n)$$

Finally it outputs the shares $(r_1, \ldots, r_n)$.

Reconstruct: The $\mathsf{RepSS}.\textsc{Reconstruct}$ algorithm takes a sharing $(r_1, \ldots, r_n)$, extract the additive sharing $(a_1, \ldots, a_n)$ and outputs

$$s = \mathsf{ASS}.\textsc{Reconstruct}(a_1, \ldots, a_n)$$

c) Verifiable Secret Sharing.: The verifiable secret sharing scheme for n parties consist of two protocols $\mathrm{VSS} = (\textsc{Share}, \textsc{Reconstruct})$, which are defined as follows:

Sharing: The $\mathrm{VSS}.\textsc{Share}$ protocol proceeds as follows

[2]Secrecy structure: the collection of ignorant party subsets, i.e., it contains all subsets of the parties that cannot learn anything about the secret.

[3]However, this can easily be extended to corruption of more parties by adapting the secret sharing schemes to the corresponding secrecy structure. See [25] for details.

1) The dealer shares their secret $s \in \mathbb{Z}_p$ using the replicated sharing algorithm

$$(r_1, \ldots, r_n) \leftarrow \mathsf{RepSS.SHARE}(s)$$

i.e., party P_i receives the share $r_i = (a_1, \ldots, a_{i-1}, \perp, a_{i+1}, \ldots, a_n)$.
2) For each $i \in [n]$, each pair of parties in $\mathcal{P} \setminus \{P_i\}$ check whether they received the same value for a_i. If any inconsistency is detected, the players broadcast a complaint.
3) The dealer broadcasts all the shares for which a majority of parties raised a complaint. The other parties accept these broadcasted values. If the dealer refuses to broadcast any of the requested shares, the protocol aborts.

Reconstruct: The VSS.RECONSTRUCT proceeds as follows

1) Party P_i knows share r_i for all $i \in [n]$.
2) All parties send their share to all other parties such that each party knows $r_1, \ldots, r_n$.
3) Now each party obtained $n-1$ copies of a_i (the underlying additive share). Each party locally does a majority vote and takes the value that occurs more than half of the time.
4) After the majority vote each party knows $a_1, \ldots, a_n$ and can reconstruct the secret using additive secret sharing

$$s = \mathsf{ASS.RECONSTRUCT}(a_1, \ldots, a_n)$$

1) Properties: All three secret sharing schemes in this section have the following properties:

a) Correctness: Let $s \in \mathbb{Z}_p$ be a secret. Then $\text{RECONSTRUCT}(\text{SHARE}(s)) = s$.

b) Secrecy: Let s_1 and s_2 be secret sharings of two secrets:

$$(a_1^1, \ldots, a_n^1) \leftarrow \text{SHARE}(s_1), \quad (a_1^2, \ldots, a_n^2) \leftarrow \text{SHARE}(s_2),$$

where party P_i knows shares a_i^1 and a_i^2. Then for all i, $a_i^1 \sim^p a_i^2$.

Both additive and replicated secret sharing schemes can be seen as passively secure secret sharing schemes as they only provide correctness guarantees against an honest dealer. Verifiable secret sharings on the other hand ensures that the dealer's sharing is consistent among parties and hence guarantees correctness against a malicious dealer.

2) Linearity of the Secret Sharing Schemes: The three secret sharing schemes in this section are all linear secret sharing schemes. Let s_1 and s_2 be secret sharings of two secrets:

$$(a_1^1, \ldots, a_n^1) \leftarrow \text{SHARE}(s_1), \quad (a_1^2, \ldots, a_n^2) \leftarrow \text{SHARE}(s_2),$$

where party P_i knows shares a_i^1 and a_i^2. The parties can compute the secret sharing of the sum $s_1 + s_2$ by performing a linear operation on the shares that they know. E.g. for additive secret sharing this operation is the addition of the known shares:

$$rs_i = a_i^1 + a_i^2 \qquad \text{for } i \in [n].$$

Then $s_1 + s_2 = \text{RECONSTRUCT}(rs_1, \ldots, rs_n)$.

III. MULTIPARTY COMPUTATION

In multi-party computation, n parties wish to compute a deterministic function f of their secret inputs. For the rest of this work, we fix a (deterministic) function $f : X_1 \times \cdots \times X_n \to Y$, and hence assume for simplicity that all parties obtain the same output[4]. The goal is to compute $y = f(\mathbf{x})$, while ensuring that the following security requirements are fulfilled:

- Correctness: The correct value of the output y is computed.
- Privacy: The output y is the only new information that can be derived from the computation.

To achieve this goal, the parties use a probabilistic protocol π to compute y.

a) The adversarial model.: When talking about security of a multi-party computation protocol, we need to consider the power of the adversary. In our setting, we consider static corruption, where the corrupted parties are determined before the protocol execution. Furthermore, we consider two types of adversarial behavior: *passive* and *active* corruption, which specify the actions the corrupted parties are allowed to take.

- In the passive model, the adversary is assumed to follow the protocol description, but attempts to learn more than the output from the protocol execution, i.e., tries to learn information that should remain private. This is also known as security against semi-honest or honest-but-curious adversaries.
- In the active model, the adversary is allowed to deviate from the protocol description arbitrarily. This is also known as security against malicious adversaries.

Moreover, we assume that the adversary is *stateful*, i.e. the adversary can maintain an internal state to log the transcript of the protocol etc.

A. The Protocol

We consider the MPC protocol presented by Maurer [25] which provides active security under corruption of $t < n/3$ parties. The protocol is based on verifiable secret sharing and computes a public function of the n parties' inputs that is represented as arithmetic circuit. The protocol consists of three phases: An input phase, where each of the n parties' secret inputs is shared using verifiable secret sharing; the computation phase, where the parties perform computations on the shares; and finally, the output phase, where each party opens their share of the result, thus allowing everyone to locally reconstruct the result. Evaluating an arithmetic circuit requires additions and multiplications. Like in many other contexts, multiplication is significantly harder to achieve than addition.

[4]Different outputs for the parties can be obtained by opening outputs only to certain parties.

In our case, the linearity of the secret sharing scheme provides a way of locally adding secret sharings, while multiplication requires further communication. For simplicity, we focus on the case of corruption of a single party.

The protocol assumes synchronous communication and the existence of private authenticated communication channels between parties as well as an authenticated broadcast channel. Even though the broadcast channel can be simulated, we will use this abstraction to ease readability. Furthermore, the adversary is allowed to be rushing, i.e. in each communication step, the adversary can see all the communication from all honest parties before sending its own.

We will now describe the phases of the protocol:

Input: Each party performs a verifiable secret sharing of their secret input (as described in Section II-A). These shares are distributed s.t. party P_j receives the j'th shares of each secret. At the end of the phase, each party is committed to an input, i.e. this input is uniquely determined from the other parties' combined shares. The following matrix represents the knowledge of one party. Each row i stands for the share received from party P_i.

$$\begin{bmatrix} a_1^1 & a_2^1 & \cdots & \perp & \cdots & a_n^1 \\ a_1^2 & a_2^2 & \cdots & \perp & \cdots & a_n^2 \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_1^n & a_2^n & \cdots & \perp & \cdots & a_n^n \end{bmatrix}$$

Computation: The parties can now compute on shared values:

Addition: Since the secret sharing scheme is linear, the parties can compute a secret sharing of the sum of any number of shared values by locally adding their shares. E.g. given n shares of values represented as matrix of shares, party P_i will collapse each column by adding the values:

$$ss_j = a_j^1 + a_j^2 + \cdots + a_j^n \qquad \text{for } j \in [n] \text{ s.t. } j \neq i$$

This provides party P_i with the i'th share of the sum of the secret inputs

$$rs_i = (ss_1, \ldots, ss_{i-1}, \perp, ss_{i+1}, \ldots, ss_n).$$

Multiplication: In order to multiply two secret shared values, the parties need to communicate and to introduce fresh randomness. Each party P_i knows the i-th share of two shared values a and b:

$$\begin{bmatrix} a_1 & a_2 & \cdots & \perp & \cdots & a_n \\ b_1 & b_2 & \cdots & \perp & \cdots & b_n \end{bmatrix}$$

Note now that $a \cdot b = \sum_{i,j} a_i b_j$, meaning that if each party knows a secret sharing of each term $a_i b_j$, then they can compute $a \cdot b$ by (locally) adding the sharings of all terms.

The parties proceed as follows to compute a secret sharing of term $a_i b_j$: First, each of the $n-2$ (or $n-1$ if $i = j$) parties that knows both a_i and b_j will compute a fresh secret sharing of $a_i b_j$. In a next step, the parties check if all sharings are sharings of the same value by first computing the pairwise differences between two sharings and then opening and reconstructing the difference. If all differences are 0, then the parties choose the sharing of an arbitrary party as sharing of $a_i b_j$. If any of the opened differences is different from 0, then the parties will compute a secret sharing of $a_i b_j$ in the following way: Each party that knows a_i reports their value of a_i to all other parties. Each party sets a_i to be the majority over all received values for a_i. The parties do the same for b_j. Then, each party sets the additive sharing of the term $a_i b_j$ to be $(a_i b_j, 0, \ldots, 0)$, computes the corresponding replicated secret sharing, and stores its share of it.

Output: Each party opens their share rs_i of the result by sending it to all other parties. This allows everyone to locally reconstruct the sum y of the secret inputs:

$$y = \text{VSS.RECONSTRUCT}(rs_1, \ldots, rs_n).$$

For simplicity, we will consider two kinds of protocol, with a single addition and multiplication, resp., as computation phase. They will be referred to as addition and multiplication protocol, respectively.

B. Passive Security

This section presents a definition of passive security against one corrupt party following the simulation paradigm, the current standard approach for defining security of MPC protocols [18].

In the setting of passive security, the adversary must follow the protocol description (i.e. the adversary must use his designated input and must send the correct messages). This might seem like a very weak security model, since it does not capture even small deviations from the protocol description. However, it guarantees that the protocol does not leak any information inadvertently (i.e. that an honest-but-curious adversary does not learn unwanted information from the transcript of the protocol). Another way of interpreting passive security is that it provides security against corruption after the execution of the protocol, i.e. what honest parties learned and stored during the protocol execution does not leak information.

The goal is to ensure that the adversary only learns the output of the computation, which means that everything it sees during the execution can be computed based on its input and the output. This property is proved by building a simulator. This simulator is given the input and output of the corrupt party and computes a view (or transcript) that is indistinguishable from the adversary's view in the real execution of the protocol.

Definition 2 (View and output)**.** *Let π be a protocol for computing f. We define the view of a party as all the messages sent and received by this party during the protocol execution, i.e., we can denote the view of party P_i by*

$$\mathsf{view}_i^{\pi}(\mathbf{x}) \stackrel{def}{=} (\mathbf{x}_i, m_i^{(1)}, \ldots, m_i^{(t)}),$$

where $m_i^{(j)}$ denote the jth message sent or received by party P_i. Furthermore, we denote the common output of all parties by

$$\mathsf{output}^{\pi}(\mathbf{x}).$$

Note that we are considering the setting of deterministic functions (like addition and multiplication) and corruption of a single party. This means that for passive security, we can consider the correctness and privacy separately [20, Chapter 2].

Definition 3 (Perfect Passive Simulation-based Security, [20])**.** *We say that an n-party protocol π securely computes f in the presence of static semi-honest adversaries, if*

Correctness: *For every $\mathbf{x} \in X_1 \times \cdots \times X_n$,*

$$\mathbb{P}[\mathsf{output}^{\pi}(\mathbf{x}) = f(\mathbf{x})] = 1.$$

Privacy: *For all $i \in [n]$, and for all adversaries that passive corrupts party P_i, there exists a polynomial-time simulator $\mathcal{S}_i$ such that*

$$\{\mathcal{S}_i(\mathbf{x}_i, f(\mathbf{x}))\}_{\mathbf{x}} \sim^p \{\mathsf{view}_i^{\pi}(\mathbf{x})\}_{\mathbf{x}}$$

where $\mathbf{x} \in X_1 \times \cdots \times X_n$.

C. Active Security

In this section, we discuss the existing cryptographic definition of active security for MPC based on the ideal-real world model. We present a natural extension of the two-party definition by Hazay and Lindell [20] to the case of n parties.[5]

Active security models the setting where a malicious (or actively corrupt) party may follow any strategy (including arbitrarily deviating from the protocol description). Thus, it is insufficient to consider the adversary's view in the protocol based on its input and output (like in the passive case). This is especially important given that the adversary may try to change its input during the protocol execution, make the output be incorrectly distributed, or make the honest parties output different or incorrect values (to mention a few possible strategies).

To capture these threats, we consider the ideal-real world model, where we compare the real execution of the protocol with an ideal execution that is secure by definition. In the ideal execution, the computation is performed by a trusted party T. This party is incorruptible and acts as a black-box such that no one can observe or influence the computation performed by this trusted party. This means that the only "attack" we allow the adversary to perform is input substitution, i.e., the only thing the adversary is able to do is to change its "given" input to something else while this new input cannot depend on the inputs of the honest parties (since the adversary in the ideal execution receives no information before having to commit to its input). Intuitively, security is shown by providing a simulator (with access to the real-world adversary) that when interacting with T in the ideal world will produce the same output distribution as in the real world. Note that this definition captures both correctness and privacy of a protocol: Privacy follows from the simulation paradigm and correctness from the fact that the ideal model always outputs the correct result.

We remark that we consider the setting of one corrupt party to match the setting used in the rest of the paper. However, the definition can easily be extended to a setting of several corrupted parties. In the information-theoretic setting that we consider in this work, protocols can achieve active security under corruption of at most $t < n/2$ parties assuming broadcast (depending on the protocol) [27]. However, Maurer's protocol allows only for $t < n/3$, even with broadcast [25].

Like in the setting of passive security, our definitions work for both polynomially bounded and unbounded simulators.

a) Notation.: Let $P_1, \ldots, P_n$ be n parties, and let $\mathcal{A}$ denote the adversary that decides to corrupt party P_a with $a \in [n]$. Let T be a trusted party that on inputs $\mathbf{x}$ correctly computes $f(\mathbf{x})$.

b) Ideal Model.: In this model each party sends their input to a trusted party T that computes the function f and returns the result to the parties.

Input: Let x_i be the input of party P_i for $i \in [n]$.
Send inputs: The honest parties P_i for $i \in [n] \setminus \{a\}$ send their inputs x_i to the trusted party T. The corrupt party P_a sends its prescribed input x_a, some other input, or a special abort symbol $\perp$ to the trusted party T. This decision is made by the adversary $\mathcal{A}$ and may depend on x_a and the adversary's internal state.
Let $\mathbf{x}'$ denote the inputs that T receives, where $\mathbf{x}'_i = \mathbf{x}_i$ for all $i \neq a$.
Receive outputs: If $\mathbf{x}'_i = \perp$ (the abort symbol) for some $i \in [n]$, then T informs all parties that the protocol aborts by sending $\perp$ to all parties. Otherwise, the trusted party T computes $y = f(\mathbf{x}')$, and sends y to all parties.
Output: The honest parties output y, while the adversary $\mathcal{A}$ outputs an arbitrary function g of the prescribed input x_a of the corrupts party and the value y obtained from the trusted party (i.e $g(x_a, y)$).

Let $\mathsf{IDEAL}_{f,\mathcal{A},a}(\mathbf{x})$ denote the output of the ideal execution. This is an n-tuple containing the honest parties' outputs y and the output of the adversary $\mathcal{A}$ from the above ideal execution

$$\mathsf{IDEAL}_{f,\mathcal{A},a}(\mathbf{x}) := (y_1, \ldots, y_n)$$

where $y_a = g(x_a, y)$ and $y_i = y$ for $i \in [n] \setminus \{a\}$

c) Real Model.: A real execution of the protocol π (with no trusted party). In this case, the adversary $\mathcal{A}$ sends all messages on behalf of the corrupt party P_a and may follow an arbitrary strategy. The honest parties must follow the protocol description.

Let $\mathsf{REAL}_{\pi,\mathcal{A},a}(\mathbf{x})$ denote the output tuple containing the honest parties' outputs and the output of the adversary $\mathcal{A}$ from the real execution of π.

Definition 4 (Perfect Active Simulation-based Security, [20])**.** *Let π be a n-party protocol that computes f. Protocol π is*

[5]Note that this extension is restricted to our setting of perfect security in the case of one corrupted party. However, this can be extended to corruption of several parties.

said to securely compute f in the presence of static malicious adversaries, if for every adversary $\mathcal{A}$ for the real model, there exists an polynomial-time adversary $\mathcal{S}$ (called a simulator) for the ideal model, such that for every $a \in [n]$

$$\{\mathsf{IDEAL}_{f,\mathcal{S},a}(\mathbf{x})\}_{\mathbf{x}} \sim^{p} \{\mathsf{REAL}_{\pi,\mathcal{A},a}(\mathbf{x})\}_{\mathbf{x}}$$

where $\mathbf{x}_i \in X_i$ for $i \in [n]$.

d) Input extraction: Any protocol satisfying Definition 4 must allow the simulator to extract the input of corrupt parties from the messages they send (up to equivalence, i.e. inputs that lead to the same output). The reason is that in the real world, the simulator can only get the output that an adversary would see from the ideal functionality by using the adversary's input. Hence, in a protocol that outputs the correct result, it must be possible to extract the adversary's input.

IV. Modelling privacy as input-independence

In this section, we present new security definitions against both passive and active corruption where privacy is based on input-independence, and prove that these new definitions imply the simulation-based definitions from Section III. These definitions will allow us to prove active security of Maurer's protocol in EasyCrypt in Section V.

In the following sections, we redefine security using a non-interference-based strategy instead of a simulation-based definition, i.e., we compare the adversary's view in two different executions of the protocol under conditions that rule out trivial distinguishability. Input-independence then means that the adversary, given only its input and the output of computation, cannot distinguish which of all possible consistent inputs of the honest parties was used.

A. Input-independence

Input independence is the non-interference property of protocols where the view of a party in the protocol is independent of the other parties' inputs. In our setting, this property will only hold only under the side condition that the inputs of the other parties must be consistent (i.e. the two executions of the protocol must lead to the same output).

Definition 5 (Non-interference)**.** *An n-party protocol π enjoys non-interference if for all $i \in [n]$, we have that for all inputs $\mathbf{x}, \mathbf{x}' \in X_1 \times \cdots \times X_n$ related under some condition $C_i(\mathbf{x}, \mathbf{x}')$, then it holds that*

$$\{\mathsf{view}_i^{\pi}(\mathbf{x})\}_{\mathbf{x}} \sim^{p} \{\mathsf{view}_i^{\pi}(\mathbf{x}')\}_{\mathbf{x}'}.$$

The definition states that given two sets of inputs that are related under some condition, then party P_i cannot distinguish between the two executions of the protocol (i.e. party P_i's view in the two executions are indistinguishable).

Looking ahead, we will model the view of the adversary in EasyCrypt using the global state of the adversary (`glob A`). Thus, given an adversary that corrupts party P_i, we can express the non-interference property in EasyCrypt pseudocode as follows

```
equiv [ π ~ π : ={glob A} ∧ C_i ⟹ ={glob A}]
```

The condition `={glob A}` means that the global state of the adversary before the two executions of π are equal, while after the executions we have equality over the distribution of the adversary's global state.

B. Passive security

Passive security will be defined again as two properties of a protocol π: correctness and privacy. However, we define the privacy property now as an input-independence property.

Recall that $\mathsf{view}_i^{\pi}(\mathbf{x})$ denotes the view of party P_i in the execution of the protocol on inputs $\mathbf{x} = (x_1, \ldots, x_n)$, and that $\mathsf{output}^{\pi}(\mathbf{x})$ denotes the common output of all parties.

For the privacy property in this definition, we will fix the input of the corrupt party to be the same in both executions. The inputs of the honest parties are chosen consistently such that the output of the computation is the same in both executions. Then input-independence means that the view of a corrupt party is independent of the actual inputs of the honest parties as long as they lead to the same output.

Definition 6 (Perfect Passive NI-based Security)**.** *We say that an n-party protocol π securely computes f in the presence of static semi-honest adversaries if*

Correctness: *For every $\mathbf{x} \in X_1 \times \cdots \times X_n$,*

$$\mathbb{P}[\mathsf{output}^{\pi}(\mathbf{x}) = f(\mathbf{x})] = 1.$$

Privacy: *For all $i \in [n]$, and for all adversaries that passively corrupt party P_i, we have that for all inputs $\mathbf{x}$, $\mathbf{x}' \in X_1 \times \cdots \times X_n$ such that $\mathbf{x}_i = \mathbf{x}'_i$ (fixed input for the corrupt party) and $f(\mathbf{x}) = f(\mathbf{x}')$, then it holds that*

$$\{\mathsf{view}_i^{\pi}(\mathbf{x})\}_{\mathbf{x}} \sim^{p} \{\mathsf{view}_i^{\pi}(\mathbf{x}')\}_{\mathbf{x}'}$$

Equivalence between the definitions.: We will now prove that the presented non-interference-based definition is equivalent to the simulation-based definition for perfect *passive* security.

Theorem 1. *Let f be an efficiently invertible function and let π be an n-party protocol that computes f. Then π is perfect passive simulation-based secure if and only if π is perfect passive NI-based secure.*

Proof. Both definitions consist of two parts, correctness and privacy. As correctness is defined the same in both definitions, we only consider the privacy part of the definitions.

Let π have perfect passive security under the NI definition. Let P_i be the corrupt party. Then by definition, there exists inputs $\mathbf{x}, \mathbf{x}' \in X_1 \times \cdots \times X_n$ with $\mathbf{x}_i = \mathbf{x}'_i$ and $f(\mathbf{x}) = f(\mathbf{x}')$, such that the protocol execution on these inputs produce equally distributed views. Hence, a simulator with input $\mathbf{x}_i$ and $y = f(\mathbf{x})$ can invert f for fixed $\mathbf{x}_i$ to obtain inputs $x'_1, \ldots, x'_{i-1}, x'_{i+1}, \ldots, x'_n$ (with x'_j possible different from x_j for $j \neq i$) for the honest parties. Then the simulator can construct a view by simulating the protocol π on inputs $\mathbf{x}'$ with $\mathbf{x}'_i = \mathbf{x}_i$.

For the other direction, let π have perfect passive security under the simulation-based definition. Let P_i be the corrupt

party and let $\mathbf{x}, \mathbf{x}' \in X_1 \times \cdots \times X_n$ such that $\mathbf{x}_i = \mathbf{x}'_i$ and $f(\mathbf{x}) = f(\mathbf{x}')$. Then by definition, there exists a simulator S_i such that

$$\{S_i(\mathbf{x}_i, f(\mathbf{x}))\}_{\mathbf{x}} \sim^p \{\mathsf{view}_i^{\pi}(\mathbf{x})\}_{\mathbf{x}}$$

and

$$\{S_i(\mathbf{x}'_i, f(\mathbf{x}'))\}_{\mathbf{x}'} \sim^p \{\mathsf{view}_i^{\pi}(\mathbf{x}')\}_{\mathbf{x}'}.$$

Since $\mathbf{x}_i = \mathbf{x}'_i$ and $f(\mathbf{x}) = f(\mathbf{x}')$, we have

$$\{S_i(\mathbf{x}_i, f(\mathbf{x}))\}_{\mathbf{x}} \sim^p \{S_i(\mathbf{x}'_i, f(\mathbf{x}'))\}_{\mathbf{x}'}$$

and hence

$$\{\mathsf{view}_i^{\pi}(\mathbf{x})\}_{\mathbf{x}} \sim^p \{\mathsf{view}_i^{\pi}(\mathbf{x}')\}_{\mathbf{x}'}$$

by transitivity of $\sim^p$. □

Note that the class of functions f that can be considered in the previous theorem can be extended if one allows computationally unbounded simulation: Given a function f, $f(\mathbf{x})$, and $\mathbf{x}_i$, an unbounded simulator can find $x'_1, \ldots, x'_{i-1}, x'_{i+1}, \ldots, x'_n$ (with x'_j possible different from x_j for $j \neq i$) such that $f(\mathbf{x}) = f(\mathbf{x}')$.

The EasyCrypt formulation of NI-based security lemma for the passive case is

```
equiv [ π ~ π :
  ={glob A, advid}
  ∧ s{1}.[advid{1}] = s{2}.[advid{2}]
  ∧ sum s{1} = sum s{2}
  ⟹
  ={glob A} ]
```

where N denotes the number of parties in the protocol.

C. Active Security

In this section, we redefine the definition of active security as three properties that a protocol must follow, and show that these properties imply simulation-based active security. This is the definition that will be used in Section V.

A simulation-based proof of active security would proceed by considering an equivalence between π and a simulator. However, non-interference properties are more amenable to computer-aided proofs, because the two runs of the same program are structurally the same. This is especially important in the active case: Note that the simulator does not receive the adversary's secret protocol input as input, but must extract it (in our case from the communication between corrupt and honest parties). As a simulator must start to interact consistently with the adversary without knowledge of the inputs of any party or the protocol output, the protocol and the simulator will have to differ more.

EasyCrypt *does* support game hopping proofs using simulators though; see e.g. Barthe et al. [5]. However, our implementation of Maurer's multiplication protocol takes roughly 500LOC, so a game hopping proof would require many large intermediate protocols and hence much code duplication.

Let π be an n-party protocol that computes f. We assume for the rest of this work that π can be split into three phases: input, computation, and output, where the output phase consists of a single round of communication. Let π_1 denote the combined input and computation phase, and let π_2 denote the output phase such that $\pi = \pi_2 \circ \pi_1$. Let $\mathsf{view}_i^{\pi_1}(\mathbf{x})$ denote the view of party P_i in the execution of π_1, and let $\mathsf{output}^{\pi}(\mathbf{x}) = (y_1, \ldots, y_n)$ denote the output of all parties after the execution of the protocol π.

Input extraction: We define an extraction function to be used in the security definition: Let $v = \mathsf{view}_i^{\pi_1}(\mathbf{x})$ be the view of party P_i after the execution of π_1 (the input and computation phase). Then there exists a polynomial-time input extraction function

$$\mathbf{x}_i \leftarrow \mathsf{in}_i(v)$$

that takes a view of P_i after execution of π_1 and outputs party P_i's committed input (i.e. the input P_i decided to use during π_1)

Note that the extraction function in_i extracts the input that the adversary is committed to after the input phase. The adversary may start with an input and change it during the input phase. After that phase, however, the adversary is committed to an input. In Maurer's protocol, this is the case because of the shares it sent to the honest parties, i.e. the shares that the honest parties received determine the adversary's input uniquely.

Definition 7 (Perfect Active NI-based Security)**.** *Let $\pi = \pi_2 \circ \pi_1$ be a protocol that computes f.*

Protocol π is said to securely compute f in the presence of static malicious adversaries if for all $a \in [n]$ and for every adversary $\mathcal{A}$ that actively corrupts party P_a, the protocol fulfills the following properties:

Correctness: *Let $\mathbf{x} \in X_1 \times \cdots \times X_n$ be the inputs to the execution and let $v = \mathsf{view}_a^{\pi_1}(\mathbf{x})$ be the view of corrupt party P_a after the execution of π_1. Let $\mathsf{output}^{\pi}(\mathbf{x}) = (y_1, \ldots, y_n)$ be the output of the protocol π, then for all $i \in [n]$ with $i \neq a$ we have*

$$\mathbb{P}[y_i = f(\mathbf{x}')] = 1$$

where $\mathbf{x}'_a = \mathsf{in}_a(v)$ (the committed input for the corrupt party) and $\mathbf{x}'_j = \mathbf{x}_j$ for all $j \neq a$ (the honest parties inputs).

Input Independence: *For all inputs $\mathbf{x}, \mathbf{x}' \in X_1 \times \cdots \times X_n$ with $\mathbf{x}_a = \mathbf{x}'_a$ (fixed input for the corrupt party),*

$$\{\mathsf{view}_a^{\pi_1}(\mathbf{x})\}_{\mathbf{x}} \sim^p \{\mathsf{view}_a^{\pi_1}(\mathbf{x}')\}_{\mathbf{x}}.$$

Output Simulation: *Let $\mathbf{x} \in X_1 \times \cdots \times X_n$ be the inputs to the execution and let $v = \mathsf{view}_a^{\pi_1}(\mathbf{x})$ be the view of party P_a after the execution of π_1. Then let $y = f(\mathbf{x}')$, where $\mathbf{x}'_a = \mathsf{in}_a(v)$ and $\mathbf{x}'_i = \mathbf{x}_i$ for all $i \neq a$. We say that the output phase π_2 preserves privacy if the final messages $\{m_i\}_{i \neq a}$ sent by the honest parties only depend on the view v and the result y, and moreover, they can be computed efficiently. I.e. the final messages follow an efficiently samplable distribution on v and y*

$$\{m_i\}_{i \neq a} \leftarrow \mathcal{D}_{v,y}.$$

NI-based implies simulation-based security: Next, we prove that the above NI-based definition implies the standard simulation-based definition. In fact, the NI-based definition is greatly inspired by a widely used strategy to construct simulators for the specific kind of protocol we have in mind.

The runtime of the simulator that we construct in the proof depends on the runtime of input extraction and output simulation. Since both are possible in polynomial-time, then the simulator will run in polynomial-time as well.

Theorem 2. *Let $\pi = \pi_2 \circ \pi_1$ be a n-party protocol computing f. If π is perfect active NI-based secure, then there exists a simulator $\mathcal{S}$ such that π is perfect active simulation-based secure.*

Proof. We start from a real protocol execution and argue about a simulation strategy. In the simulation-based security definition, there exists a simulator that has to simulate messages sent by the honest parties without knowing their inputs. Moreover, this simulator has oracle access to a trusted party T that knows the honest parties' inputs. Therefore, the general simulation strategy is to extract the adversary's committed input from the messages he sends during π_1. Then the simulator can query the trusted party for the correct output. Given this, the simulator can compute the final messages sent by the honest parties in the output phase.

We have to argue now that this strategy is feasible given a protocol with the properties stated above as well as that the strategy is indistinguishable from a real protocol execution to the adversary. We construct the following simulator:

Simulator $\mathcal{S}$:

1) Run π_1 with the adversary while simulating the honest parties with default inputs (e.g., $\mathbf{x}_i = 0$ for all $i \neq a$). Let $v = \mathsf{view}_a^{\pi_1}(\mathbf{x})$ be the view of the corrupt party P_a after the execution of π_1 (i.e., all communication between the corrupt party and the simulator).
2) Extract the input $\mathbf{x}'_a$ that the adversary is committed to after the input phase as $\mathbf{x}'_a = \mathsf{in}_a(v)$. Send $\mathbf{x}'_a$ to the trusted party T to obtain the output y.
3) Sample the messages that the honest parties send in the output phase as $\{m_i\}_{i \neq a} \leftarrow \mathcal{D}_{v,y}$.

We can now show that a protocol execution simulated with $\mathcal{S}$ is indistinguishable from a real protocol execution. Input independence implies that the adversary's view after π_1 is independent of the honest parties' inputs. In particular, no adversary can distinguish between a view from executing π_1 with real inputs for the honest parties from one with inputs 0 for all honest parties. Hence, the adversary can not distinguish between the real execution and the simulated execution of π_1.

Then, the simulator gathers the view of the corrupt party $v = \mathsf{view}_a^{\pi_1}(\mathbf{x})$, and sends $\mathbf{x}'_a = \mathsf{in}_a(v)$ to the trusted party T. The trusted party computes and returns $y = f(\mathbf{x}')$, where $\mathbf{x}'_i = \mathbf{x}_i$ for all $i \neq a$, i.e., the real inputs of the honest parties. Thus, the simulator gets the correct output because of the correctness property of the protocol.

Finally, output simulation guarantees that the final messages the honest parties send in the output phase only depend on the view v of the corrupt party after π_1 and the correct output of the protocol y. Thus, the simulator can sample these messages $\{m_i\}_{i \neq a}$ according to the distribution $\mathcal{D}_{v,y}$. The adversary knows the final message m_a that it is supposed to send in the output phase since m_a is uniquely determined by its view in the protocol. Thus, given the messages $\{m_i\}_{i \in [n]}$, the adversary can compute the correct output y. Furthermore, the adversary learns no more than the output, since the messages sent by the simulator (on behalf of the honest parties) only depend on the output and the adversary's view after π_1 (which did not reveal any information about the honest parties inputs). □

Remark: The other direction, from simulation-based security to NI-based security, does not hold in general as the simulator may use a simulation strategy that is incompatible with the NI-definition.

V. Modelling in EasyCrypt

A. EasyCrypt

EasyCrypt is a proof assistant for verifying the security of cryptographic constructions in the computational model. EasyCrypt provides a simple imperative probabilistic programming language pWhile to specify protocols.

As an example of EasyCrypt code, consider the additive sharing protocol consisting of share and reconstruct procedures.

```
proc share_additive(s : zmod) : zmod list =
{ var mxrd;
  mxrd <$ dlist dzmod (N-1);
  return (s - sum mxrd) :: mxrd;}

proc reconstruct_additive (sx : zmod list) : zmod =
{ return sum sx; }
```

Here `mxrd <$ dlist dzmod (N-1)` samples from a uniform distribution on lists over `zmod` of size $N-1$.

Proving is done using a variety of (probabilistic relational) Hoare logics. Mathematical functions and data types are defined using an ambient higher order logic and a functional programming language. EasyCrypt has both tactic based interactive proofs, but also automatic proofs, using an SMT backend.

Modelling and proving is done in two ways. When dealing with honest parties, we tend to use functional programs, so-called operators, and use the ambient logic to reason about these programs. Adversarial code is treated using a module system and procedure calls. One specifies the module type of the adversarial code, and proves properties over all possible instances of this module type. The module system is connected to the imperative pWhile language[6], so we reason in the corresponding Hoare logic. Usually, the main effort is to find the correct pre- and post-conditions and loop invariants. Often we are arguing that the adversary is harmless in certain parts

[6]The choice for an *imperative* probabilistic language is not forced. One could also use a functional probabilistic programming language, such as Rml, instead of pWhile. Rml used in the ALEA Coq-library [2], the base for CertiCrypt [7], the predecessor of EasyCrypt. However, such a functional language is not implemented in EasyCrypt.

of the protocol. The way to specify this is via the equivalence of adversarial (imperative) code and functional code. This kind of reasoning is familiar in program correctness.

B. Modelling the Protocol

In this section we discuss how to model Maurer's MPC protocol [25] and prove active security.

a) Adversary and phases: Recall that a malicious adversary can deviate arbitrarily from the protocol, e.g., by sending wrong or malformed messages or aborting the protocol. To model these arbitrary actions, we use the abstract module types of EasyCrypt to provide an interface to the adversary, while at the same allowing it to deviate from the protocol description. Thus, for each stage in the protocol, whenever we want the adversary to do some computation, send information, or receive information, we call the adversary's abstract procedures. E.g., in the output phase, we send the honest parties' shares of the result `psums` to the adversary and ask the adversary to send his share to all other parties:

```
advc <@ A.bxshareofres(psums);
```

Note that according to the protocol description, the adversary is supposed to send its share of the result to all other parties (i.e., everyone should receive the same share). However, the adversary has the power to send different and (possibly) wrong shares to the other parties. We model this by letting the adversary return a matrix `advc`, where row i is the share that adversary sends to party P_i.

6In this setting, we can present the general structure of the three phases of the addition protocol in EasyCrypt code. Here `<@` denote a procedure call.

```
proc input(s : zmod list) : zmod matrix list = {
  var shares;

  (* Distribution of shares: *)
  shares <@ do_sharing(s);
  pshares ← distribute_shares shares;

  (* - adversary receives shares from the honest
    parties *)
  A.recv_shares(pshares.[advid]);

  (* Consistency check: *)
  (* - collect the complaints *)
  rx <@ verify_shares();

  (* - adversary logs the requests since they are
    public *)
  A.recv_rx(rx);

  (* - reply the complaints *)
  bx <@ broadcast_shares(rx);

  (* - adversary logs the broadcast values *)
  A.recv_bx(bx);

  (* - fix the parties' views from the broadcast
    values *)
  pshares ← fix advid pshares bx;

  return pshares;}

proc computation (pshares : zmod matrix list) : zmod
     matrix = {
  (* - notify the adversary about the start of the
    phase *)
  A.localsum();
  (* - perform the local addition for the honest
    parties *)
  return mklocalsums advid pshares; }

proc output (psums : zmod matrix) : zmod list = {
  var advc, advres, resshares;

  (* - the adversary receives shares of the result
   *   from the honest parties and sends his own
    share *)
  advc <@ A.bxshareofres(psums);
  resshares ←distribute_resshares advid advc psums;

  (* - get the adversary's result.. *)
  advres <@ A.getres();

  (* ... and return results of all parties *)
  return reconstruct_vss advid advres resshares; }
```

The input and output phase for the multiplication protocol are essentially the same. The computation phase looks as follows. Importantly, the computation phase contains (a lot of) communication which is modelled by matrices keeping track of all messages that were sent and received.

```
proc multiplication (a, b : zmod matrix) : zmod
    matrix = {
    var known_shared_terms, shared_terms_rep,
    sharedterms_rep_distr, opened_diff, sharedterms;

    (* distribution of shared terms a_i b_j *)
    shared_terms_rep <@ mult_term_sharing(a,b);
    sharedterms_rep_distr ← distribute_shared_terms
     shared_terms_rep;

    (* compute and open pairwise differences of term
     sharings *)
    opened_diff <@ mult_check_term_sharing (
    sharedterms_rep_distr);

    (* choose sharing for each term *)
    sharedterms <@ mult_determine_term_sharing (a,b,
    sharedterms_rep_distr, opened_diff);

    (* add all terms: *)
    (* - rearrange for convenience *)
    known_shared_terms ← rearrange_sharedterms
    sharedterms;
    (* - inform adversary of local computation *)
    A.localmultsum();
    (* - add term sharings *)
    return mklocalsums_M ((N-1)*N)
    known_shared_terms;}
```

Here `advid` is the id of the party corrupted by the adversary.

b) Communication: EasyCrypt has no native support for communication. However, the logic and the module system are rich enough to express this. We use lists and matrices to keep track of the messages that are sent. To model the communication with the adversary, we specify abstract procedures to both send and receive messages. The EasyCrypt logic keeps track of the global state of the adversary. In the example above, `pshares` is the knowledge that each party has. The various calls to `A` are used to send/receive information to/from the adversary, using EasyCrypt's stateful modules.

c) Notes on the implementation: We note that this implementation has a few limitations regarding the power of the

malicious adversary compared to the more general description by Maurer.

We consider a setting where the adversary can only abort during the input phase. Since we consider a protocol that is secure in the presence of an honest majority, the information shared by the honest parties after a successful input phase will be enough to compute the result of the protocol. This means that if the adversary aborts during the output phase, then the honest parties will still be able to reconstruct the result of the computation.

Another limitation is that in the input phase, the adversary is forced to send shares of its secret before it can see the shares of the honest parties secret. This might seem to limit the adversary's power of choosing its input based on this extra information. However, the adversary is only committed to an input at the end of the phase. Since the adversary's initial shares cannot be forced to be consistent, during the consistency check, the adversary can still change the shares of its input, which will allow to change its input to something else (that possibly depends on the information received so far).

d) Extraction: As mentioned in Section III-C and IV-C, a proof of active security requires the extraction of information from the adversary's communication. In particular, we can extract the adversary's input and share of the output it is supposed to have from the messages that he sends and receives in the input phase.

The input extractor `extract` collects all messages that the adversary sent to honest parties(in our case, all other parties) during the input phase, and uses their shares as shares of the adversary's input. These shares can now be used to reconstruct the input. The verifiable secret sharing guarantees here that all honest parties received consistent shares that when combined reconstruct to the uniquely determined value that the adversary is committed to after the input phase.

```
op extract (advid : int) (pview : zmod matrix3) =
    col N advid pview.
```

e) High level structure: The whole protocol is then as follows. `pi1` consists of the input and computation phase. `Protocol` consists of `pi1` and the output phase. For the purpose of our proofs, the protocol also calls the extractor `extract_advinpsx` and then stores an updated input list `secrets`.

```
proc pi1 (s : zmod list) : zmod matrix list * zmod
    matrix = {
  var rinp, rcomp;

  rinp  <@ input (s);
  rcomp <@ computation (rinp);

  return (rinp, rcomp); }

proc protocol(s : zmod list) : zmod list = {
  var inp, out;

  (inp, comp) <@ pi1 (s);

  (* input extraction *)
  advinpsx ← mkaddshares (extract advid inp);
  secrets ← s.[advid ← reconstruct advinpsx];

  out <@ output (comp);

  return out; }
```

C. Outline of the proof for active security

In this section, we present the key steps in proving active security of Maurer's protocol. This is done by proving that the protocol fulfills the three properties in Definition 7: correctness, input independence, and output simulation. We will present each property in EasyCrypt code and provide an overview of the proof. We use N in the implementation to denote the number of parties. Furthermore, parties are indexed from 0. The lemmas only hold as long as $N \geq 4$, since the reconstruction procedure of the verifiable secret sharing scheme takes the majority over $N-1$ values which is only well-defined if there are at least 3 values to compare. `advid` will denote the id of the adversary. The final messages that the honest parties sent in the output phase are their shares of the result, and these messages are denoted by `comp`.

Note that EasyCrypt cannot check the runtime of code. However, it is easy to see that input extraction and output simulation run in polynomial-time in our case.

1) Correctness: In order to state correctness, we first need to define the inputs to the protocol. Honest parties will use the inputs they were assigned to in the beginning. As mentioned before, the adversary may change its mind about its input, but only during the input phase. Afterwards, it is committed to a unique input. Therefore, we extracted the shares `advinpsx` of the input in `protocol`, and then define correctness with respect to the updated input list `secrets` they define.

```
lemma correctness sx : hoare [ protocol :
  sx = s ∧ size s = N ∧ 0 ≤ advid < N
  ⟹
  0 ≤ id < N ∧ id ≠ advid ⇒
    res.[id] = f secrets ].
```

For the input and output phase, correctness of the protocol steps can be reduced to correctness of the secret sharing scheme. For the computation phase, we prove that if the inputs to a gate are secret sharings of secrets $x_1, \ldots, x_t$ (t depends on the gate), then the output of the gate is a secret sharing of the gate function on inputs $x_1, \ldots, x_t$. For addition, this property follows from linearity of the secret sharing scheme. For multiplication, note that if $(a_0, \ldots, a_{N-1})$ and $(b_0, \ldots, b_{N-1})$ are the additive secret sharings corresponding to the replicated secret sharings of inputs a and b, then $a \cdot b = \sum_{i,j} a_i b_j$. Since the secret sharing scheme is linear, it is sufficient to prove that the secret sharing of any term $a_i b_j$ that the parties agree on is actually a secret sharing of the value $a_i \cdot b_j$. In the multiplication protocol, all honest parties that know a_i and b_j will output a secret sharing of the value $a_i \cdot b_j$, whereas the adversary (if involved, i.e. if it knows both a_i and b_j) may output a secret sharing of an arbitrary value. The subsequent check computes the pairwise differences between those sharings. If all opened differences are 0, then the presence of at least one honest party guarantees that all secret sharings are sharings of the correct value $a_i \cdot b_j$. Otherwise, the replicated secret sharing corresponding to the additive sharing $(a_i b_j, 0, \ldots, 0)$ is a secret sharing of $a_i \cdot b_j$.

2) Input Independence: After the execution of π_1 (i.e. the input and computation phases), we show that the knowledge (global state) of the adversary is independent of the inputs

of the honest parties, meaning that the adversary cannot distinguish between two executions of π_1 with different inputs for the honest parties.

```
lemma input_independence : equiv [ pi1 ~ pi1 :
     ={glob A, advid}
  ∧ 0 ≤ advid{1} < N
  ∧ s{1}.[advid{1}] = s{2}.[advid{2}]
     ⟹ ={glob A} ].
```

For the input phase, we prove input independence, which follows from secrecy of the secret sharing scheme, and integrity of the output, i.e. each party knows the same additive shares (except for the one they are not supposed to know) of each shared input. Addition does not involve any communication and hence preserves input independence. For multiplication, we will consider again a term sharing $a_i b_j$. If this term sharing has input independence (with respect to the honest parties' inputs into the protocol), then the sum of all term sharings will have this property as well, and so will the output of a multiplication gate. Having the parties output sharings of $a_i b_j$ preserves input independence because of secrecy of the secret sharing scheme. Furthermore, all honest parties that share a value for the current term will share the same value. When computing and opening the pairwise differences of the sharings, each difference will be 0 if and only if the sharings that were compared are sharings of the same value. In particular, if all sharings are sharings of the same value, then all opened differences will be 0 in both executions, and hence won't provide an adversary with any input-dependent knowledge. If not all differences are 0, then one of the parties (the adversary) must have shared an incorrect value, i.e. not $a_i b_j$. In this case, the adversary must have caused the opened difference to be different from 0 (and the adversary knows this difference in advance). Since the whole protocol so far was input independent, the opened differences follow the same distribution in both executions, and hence the check preserves input independence. In the final step, the parties agree on a sharing of $a_i b_j$. If all differences were 0, then the parties will use the secret sharings of one of the parties, which preserves input independence by secrecy of the secret sharing scheme. Otherwise, the parties report their values for a_i and b_j and compute a replicated secret sharing from them. Again, this case can only occur if the adversary knows both a_i and b_j, and hence this step preserves input independence.

3) Output Simulation: The messages sent by the honest parties in the beginning of the output phase will only reveal the result of the computation. To prove this, we need to show that these messages follow some distribution on the result and the view of the adversary. This is proven by giving the exact function that maps the result and the adversary's view into the messages the honest parties sent.

Here we note that these final messages are the honest parties' shares of the result, and the view of the adversary contains its share of the result. From the definition of replicated secret sharing, we notice that the adversary's share of the result contains $N-1$ of the N additive shares that sum up to the result. Thus, we can easily reconstruct the missing value, and construct the honest parties replicated shares of the result.

```
op finalmsg (advid : int) (y : zmod) (pviewadv :
     zmod list) : zmod matrix =
  mkseq (fun i ⇒
  mkseq (fun j ⇒
    if i = advid then ⊥
    else if i = j then ⊥
    else if j = advid then y
         - sum (drop_elem advid pviewadv)
    else pviewadv.[j]) N) N.

lemma output_simulation : hoare [ protocol :
  size s = N ∧ 0 ≤ advid < N
  ⟹  finalmsg advid y advresshares = comp ].
```

D. The formalization

We have demonstrated how to formalize active security of MPC protocols in EasyCrypt. We have done this by an implementation of Maurer's MPC protocol in EasyCrypt. For the addition protocol, we have a complete proof for the three properties correctness, output simulation and input independence that are required by our non-interference based security definition. We have extended this to the multiplication protocol by identifying and formalizing all the invariants for both privacy and correctness of the multiplication protocol, and we have proved correctness, output simulation and input independence. We have permitted ourselves the license not to reprove some parts of the multiplication protocol that were very similar to the addition protocol. The statements and proof structure are very similar. However, refactoring all our code to convince EasyCrypt of this fact does not seem to provide enough insight to merit this effort. All these places are clearly documented and marked with `admit` in the sources.

Our formalization is substantial: as a very rough measure, it consists of approximately 5000LOC, 1800 of which are used for the addition protocol. The code is dense, as it combines the efficient ssreflect language with SMT-calls (for comparison, the easycrypt standard library consists of 18000LOC).

More generally, we have given a methodology to attack complex simulation-based proofs of protocols involving much communication. We did this by translating simulation-based proofs to NI-arguments and using the EasyCrypt module system to model arbitrary adversarial code. Simulation-based proofs are a standard technique in cryptography, but they are difficult to make mathematically precise. A good example is the ongoing effort to make the simulation-based UC framework completely precise, e.g. discussed in [15]. The framework is in general well understood, implementation in a proof assistant is still lacking; see [13].

VI. Related work

EasyCrypt is a specialized proof assistant for security proofs. To our knowledge, EasyCrypt is the only tool that currently allows us to conveniently verify MPC protocols in the manner that we did, since it combines a rich ambient logic with an embedded logic for a probabilistic programming language. However, EasyCrypt grew out of a Coq library [7] and, in principle the techniques we present here could also be

used in other proof assistants for higher order logic, or type theory, once one defines the programming logic in the ambient language; see e.g. [21] for a framework that supports imperative, but non-probabilistic, program logics in Coq. The general purpose proof assistant Coq has been used to verify crypto protocols in the foundational cryptography framework [26] and in the verification of an OpenSSL implementation of HMAC [9]. A similar library could be built in F*, as suggested in [10]. This would have the added benefit of a build-in SMT solver.

Wysteria is a domain specific language for MPC. It has been embedded [28] in the F^* programming language/proof assistant. Various protocols have been verified in Wysteria* to be secure against a *passive* adversary. However, all the cryptographic primitives are treated axiomatically as F^* does not include probabilistic computation. In this paper, we treat all the aspects, consider *active* security and treat the multiplication protocol.

A simulation-based proof for two parties has been formalized in the Isabelle proof assistant [14], based on CryptHOL [24]. The logic of Isabelle is similar to EasyCrypts ambient logic. However, Isabelle lacks built-in pHoare logics. They use a shallow embedding of a probabilistic programming language into Isabelle using a monadic interpretation. This is less powerful than the deep embedding used in Certicrypt [7], and implicitly in EasyCrypt. In Isabelle, they prove security against a passive (semi-honest) adversary of a two-party multiplication protocol using simulation based proofs. In contrast, we prove security of a much more complicated protocol that is secure against an active adversary and works for n-parties. This requires us to model the adversary abstractly using EasyCrypts modules system, and we thus have the harder job to reason about imperative code. Importantly, we also provide new proof techniques that are more amenable to automation, as they are close to the proof techniques used for program logics.

CryptoVerif [12] is an automatic protocol verifier. It targets different goals than EasyCrypt and is complementary to it. Indeed, CryptoVerif aims to automate cryptographic game transformations. It applies a collection of game transformations, using a full automatic proof strategy that can be driven by users' hints. On the other hand, EasyCrypt, relies on an embedding of Probabilistic Relational Hoare Logic that subsumes cryptographic games transformations. Its logic being relatively complete, it can be used to prove various properties about a large class of cryptographic primitives. Although we have not tried it, we are very doubtful that Maurer's proof can be reconstructed automatically.

VII. Conclusions and future work

A. Easycrypt

The EasyCrypt logic and module system were a good fit to express these protocols in a natural way, once we found the right way of modelling it.

In our experience, developing in EasyCrypt is fairly pleasant. The combination of interactive theorem proving using the ssreflect language combined with automatic theorem proving (SMT) is very powerful. Unfortunately, at the moment the SMT-solver does not provide information which lemmas were used unlike e.g. [22]. This information could be used to speed up checking the document (which currently takes a couple of minutes), but also be used to prove similar lemmas. Many uses of SMT could be avoided by using a more expressive type system. In particular, good support for (coersive) subtyping would have been helpful. Since much of the low-level proving is automated, leaving out simple type information will often result in the SMT-solver failing without further information. In this case, it would be helpful to provide a counterexample to the user. This functionality is provided by a number of SMT-solvers and also by quickcheck. EasyCrypt's pWhile language does not support iterators (for-loops). Most of our constructions are iterations over the list of parties. Our first modelling consisted of growing lists by a while loop. We have found it is more convenient to start the loop with an array with default values and update during the while loop.

Finally, we spent much time on the whiteboard trying to connect the code for modelling communication with our visual representation of matrices. A simple evaluator for functional programs would have been useful.

B. Future work

We have proved the security against one corrupted party. The same methodology works for more corrupted parties, as we can just give more information to the adversary, using EasyCrypt's module system. We would make a predicate `Honest` on the interval $[0..N)$. Our current formalization checks if the current party `i` equals `advid`. Instead we would check `Honest i`. Currently only the information of one party is sent to the adversary (an EasyCrypt module). Instead, we would send the information of all the corrupted parties to the adversary. Like in the current protocol, all the corrupted parties would move last.

It would be interesting to code arithmetical circuits in EasyCrypt. This would be an effort similar to the encoding of multivariate polynomials in Coq (https://github.com/math-comp/multinomials).

From a higher perspective, it would be very interesting to formally connect our work with an efficient implementation, as is done in high-assurance crypto [1].

C. Conclusion

We have presented new security definitions for active security of MPC protocols, shown that they imply the standard ones and formalized the security proof in EasyCrypt. This is the first formalized protocol with a proof of active security and the first one for n-parties.

Acknowledgements

Gilles Barthe showed us how non-interference can be used in the context of MPC for a passive adversary. Ivan Damgård helped us to understand MPC protocols and their security proofs. In the beginning of the project we profitted from

discussions with Aslan Askarov, Michael Nielsen, and Mathias Pedersen. We are grateful to all of them.

Helene Haagh and Sabine Oechsner were supported by the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme under grant agreement No 669255 (MPCPRO), the Danish Independent Research Council under Grant-ID DFF-6108-00169 (FoCC), and the European Union's Horizon 2020 research and innovation programme under grant agreement No 731583 (SODA). Bas Spitters was supported by the Guarded Homotopy Type Theory project, funded by the Villum Foundation, project number 12386.

REFERENCES

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Guillaume Davy, François Dupressoir, Benjamin Grégoire, and Pierre-Yves Strub. Verified implementations for secure and verifiable computation. *IACR Cryptology ePrint Archive*, 2014:456, 2014.

[2] Philippe Audebaud and Christine Paulin-Mohring. Proofs of randomized algorithms in coq. *Sci. Comput. Program.*, 74(8):568–589, 2009.

[3] Michael Backes and Birgit Pfitzmann. Computational probabilistic noninterference. *Int. J. Inf. Sec.*, 3(1):42–60, 2004.

[4] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 116–129. ACM, 2016.

[5] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In Alessandro Aldini, Javier Lopez, and Fabio Martinelli, editors, *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2013.

[6] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011: 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, pages 71–90, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[7] Santiago Zanella Béguelin. *Formal certification of game-based cryptographic proofs. (Certification formelle de preuves cryptographiques basées sur les séquences de jeux)*. PhD thesis, Mines ParisTech, France, 2010.

[8] Mihir Bellare and Phillip Rogaway. Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Report 2004/331, 2004.

[9] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of openssl HMAC. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 207–221. USENIX Association, 2015.

[10] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Catalin Hritcu, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Peng Wang, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. Verified low-level programming embedded in F*. arXiv:1703.00053, February 2017.

[11] George Robert Blakley. Safeguarding cryptographic keys. *Proc. of the National Computer Conference 1979*, 48:313–317, 1979.

[12] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1–2):1–135, October 2016.

[13] Chris Brzuska, Antoine Delignat-Lavaud, Konrad Kohbrok, and Markulf Kohlweiss. State-separating proofs: A reduction methodology for real-world protocols. Cryptology ePrint Archive, Report 2018/306, 2018. https://eprint.iacr.org/2018/306.

[14] David Butler, David Aspinall, and Adrià Gascón. How to simulate it in isabelle: Towards formal proof for secure multi-party computation. In *International Conference on Interactive Theorem Proving*, pages 114–130. Springer, 2017.

[15] Ran Canetti, Asaf Cohen, and Yehuda Lindell. A simpler variant of universally composable security for standard multiparty computation. In *Annual Cryptology Conference*, pages 3–22. Springer, 2015.

[16] Stephen Checkoway, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J Bernstein, Jake Maskiewicz, Hovav Shacham, Matthew Fredrikson, et al. On the practical exploitability of dual ec in tls implementations. In *USENIX security symposium*, pages 319–335, 2014.

[17] Ronald Cramer, Ivan Bjerre Damgaard, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, New York, NY, USA, 1st edition, 2015.

[18] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229, 1987.

[19] Thomas C Hales. The NSA back door to NIST. *Notices of the AMS*, 61(2):190–19, 2013.

[20] Carmit Hazay and Yehuda Lindell. *Efficient Secure Two-Party Protocols: Techniques and Constructions*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.

[21] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic.(2017). *Submitted for publication*, 1, 2017.

[22] Cezary Kaliszyk and Josef Urban. Hol(y)hammer: Online ATP service for HOL light. *Mathematics in Computer Science*, 9(1):5–22, 2015.

[23] John Launchbury, Dave Archer, Thomas DuBuisson, and Eric Mertens. Application-scale secure multiparty computation. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 8–26. Springer, 2014.

[24] Andreas Lochbihler. Probabilistic functions and cryptographic oracles in higher order logic. In Peter Thiemann, editor, *Programming Languages and Systems*, pages 503–531. Springer, 2016.

[25] Ueli Maurer. Secure multi-party computation made simple. *Discrete Applied Mathematics*, 154(2):370 – 381, 2006. Coding and Cryptography.

[26] Adam Petcher and Greg Morrisett. The foundational cryptography framework. In Riccardo Focardi and Andrew C. Myers, editors, *Principles of Security and Trust - 4th International Conference, POST 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*, volume 9036 of *Lecture Notes in Computer Science*, pages 53–72. Springer, 2015.

[27] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In David S. Johnson, editor, *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*, pages 73–85. ACM, 1989.

[28] Aseem Rastogi, Nikhil Swamy, and Michael Hicks. Wys*: A verified language extension for secure multi-party computations, August 2016.

[29] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[30] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167. IEEE Computer Society, 1986.

[31] Jean Karim Zinzindohoue, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan. A verified extensible library of elliptic curves. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 296–309. IEEE Computer Society, 2016.