A High-Assurance Evaluator for Machine-Checked Secure Multiparty Computation

Karim Eldefrawy[1] and Vitor Pereira[2,3]

[1] SRI International
[2] INESC TEC
[3] FC Universidade do Porto

Abstract. Secure Multiparty Computation (MPC) enables a group of n distrusting parties to jointly compute a function using private inputs. MPC guarantees correctness of computation and confidentiality of inputs if no more than a threshold t of the parties are corrupted. Proactive MPC (PMPC) addresses the stronger threat model of a *mobile adversary* that controls a changing set of parties (but only up to t at any instant), and may eventually corrupt *all n parties* over a long time.

This paper takes a first stab at developing high-assurance implementations of (P)MPC. We formalize in EasyCrypt, a tool-assisted framework for building high-confidence cryptographic proofs, several abstract and reusable variations of secret sharing and of (P)MPC protocols building on them. Using those, we prove a series of abstract theorems for the proactive setting. We implement and perform computer-checked security proofs of concrete instantiations of the required (abstract) protocols in EasyCrypt.

We also develop a new tool-chain to extract high-assurance executable implementations of protocols formalized and verified in EasyCrypt. Our tool-chain uses Why3 as an intermediate tool, and enables us to extract executable code from our (P)MPC formalizations. We conduct an evaluation of the extracted executables by comparing their performance to performance of manually implemented versions using Python-based Charm framework for prototyping cryptographic schemes. We argue that the small overhead of our high-assurance executables is a reasonable price to pay for the increased confidence about their correctness and security.

Keywords: Secure Multiparty Computation, Verified Implementation, High-Assurance Cryptography

1 Introduction

Correctly designing secure cryptographic primitives and protocols is a non-trivial task. We argue that not only is it hard to correctly design them, but so is it to correctly and securely implement them in software. This issue is particularly amplified in settings where protocols involve multiple parties, and when they should guarantee security against strong adversaries beyond passive ones[4].

There are several efforts implementing (advanced) cryptographic primitives and protocols available to developers, such as OpenSSL [5], s2n [6], BouncyCastle [7], Charm [2], SCAPI [34], FRESCO [8] [30], TASTY [49], SCALE-MAMBA [9], EMP [10], Sharemind [22,23]. Such tools and libraries typically aim to improve usability, software reliability, and performance. Some of them also target cloud-based and large distributed applications, aiming to deploy secure and privacy-preserving distributed computations to address practical challenges. A missing aspect of most of these efforts is the increased confidence obtained in security and correctness of

[4] Often called honest-but-curious or semi-honest, we use those terms interchangeably in this paper.
[5] https://www.openssl.org
[6] https://github.com/awslabs/s2n
[7] https://www.bouncycastle.org
[8] https://github.com/aicis/fresco
[9] `https://github.com/KULeuven-COSIC/SCALE-MAMBA`
[10] `https://github.com/emp-toolkit`

the design and implementation of such complex (cryptographic) algorithms and protocols when utilizing computer-aided formal verification and synthesis. Because of this, subsequent work [3,14,20,19,24,41,10,4] started tackling verification of cryptographic primitives and protocols, and software implementations thereof. One notable example is Project Everest [11] which provides a collection of tools and libraries that can be combined together and generate a mixture of C and assembly code that implements TLS 1.3, with proofs of safety, correctness, security and various forms of side-channel resistance.

Several authors began exploring how such high-assurance design and implementation of cryptography can be applied to more complex secure (two-party) computation and multiparty computation (MPC) protocols [9,29,23,56,64,28,46,4]. Nevertheless, we identify a series of limitations of such efforts:

1) *Limited number of parties* - [29,4,56] such work typically considers the case of two-party computation.
2) *Lack of security guarantees against strong adversaries* - the work that goes beyond two parties [9,23,64,68,47] typically focuses on the semi-honest adversary model.
3) *Lack of high-assurance implementations for active adversaries* - even when active adversaries and multiple parties are considered [46,47], there are no automatically (and verifiably) synthesized executable implementations from protocol specifications that were checked using computer-aided verification.

Table 1 summarizes the most relevant recent work on verification of complex cryptographic protocols involving multiple parties and withstanding strong adversaries. A more detailed comparison with related work is provided in Section 6.

Finally, the semi-honest and active adversary models, while covering a large set of possible applications of MPC, do not cover scenarios in which complex distributed systems are built and run for long durations, and where strong persistent adversaries continuously attack them. For example, those two models do not cover the case where adversaries can corrupt all parties over a long period of time. Proactive MPC (PMPC) addresses the stronger threat model of a *mobile adversary* that controls a changing set of parties (but only up to t at any instant), and may eventually corrupt *all n parties* over a long time throughout the course of a protocol's execution, or lifetime of confidential inputs. The main intuition behind proactive secret sharing (that typically underlies PMPC) is to periodically re-randomize (*refresh*) shares of secrets and delete old ones, thus preventing an adversary that collects all shares from different periods to combine them together. In the proactive setting, parties are periodically reset (*recovered*) to a clean state to ensure that adversarial corruptions are purged. Such reset parties have to run a recovery protocol to obtain their new non-corrupted state (secret shares) and re-join the secure computation system.

To the best of our knowledge there are currently no publicly available high-assurance implementations of formally verified and machine-checked (P)MPC withstanding active adversaries. By high-assurance we mean automatically (and verifiably) synthesized from protocol specifications that were checked using computer-aided verification. This paper takes a first stab at developing such high-assurance implementations of (P)MPC, and as a side contribution performs the first computer-aided verification of a variant of the fundamental BGW [18] MPC protocol for passive and static active adversaries using EasyCrypt (with computational security in the latter case).

Below we discuss challenges facing our work, followed by details of our contributions addressing these challenges. We finish this section with a brief description of our final verified (P)MPC evaluator (illustrated in Figure 1).

1.1 Challenges

We face two main challenges: developing machine-checked specifications and proofs for (P)MPC and underlying building block primitives, and obtaining automatically synthesized verified implementations of such protocols.

Machine-checked formalization and security proofs of MPC protocols (with multiple parties, sub-protocols, and guaranteeing security against mobile active adversaries) is a complex task that involves knowledge that spans cryptography, distributed computing and programming languages. Particularly, using a tool

[11] `https://project-everest.github.io`

Previous Work	Protocol / Num. Parties	Adversary Model	E2E Proof	High-Assurance
[68]	PCR [68] / 3 Parties	Passive	Yes	No
[4]	Yao [72] / 2 Parties	Passive	Yes	Yes
[47]	Maurer [58] / N Parties	Active	No*	No
Our Work	*BGW [18] / N Parties*	*Passive & (Pro)Active*	*Yes*	*Yes*

Table 1: Comparison of the most relevant computer-aided verification and high-assurance implementations of secure computation protocols. Private Count Retrieval (PCR) is an application-specific database querying protocol involving 3 parties (client, server, and a trusted third party). *The authors in [47] prove that certain properties are satisfied by the protocol in [58] is checked in EasyCrypt, then it is manually proven that a simulator exists if these properties hold. Our work contains E2E proofs in EasyCrypt without any manual steps.

like EasyCrypt to come up with a machine-checked proof is not easy, because one has to accommodate complex cryptographic schemes and protocol using descriptions that can be used inside the EasyCrypt system. Additionally, formalizing MPC protocols also involves formalizing the underlying mathematical structures upon which they build. Despite EasyCrypt already providing a large set of mathematical constructions, we still needed to formalize additional libraries, specially to deal with polynomials over finite fields.

There are currently no (publicly available) tools for automatically synthesizing verified implementations of complex cryptographic (multiparty) protocols. Two possible approaches to tackle this objective consist on either starting with a software implementation of the protocol and then attempt to prove properties surrounding that implementation or starting with a proof script (with a formal specification and proofs) and derive a concrete implementation from it. In this work, we generated a concrete verified *correct-by-construction* executable software implementation from a proof script. Using our new EasyCrypt extraction tool-chain, we were able to obtain such implementation of an MPC evaluator with (optional) proactive security guarantees.

1.2 Contributions

In this paper, *we develop a high-assurance formally-verified proactive secure multiparty computation, (P)MPC, evaluator based on machine-checked proactive (verifiable) secret sharing and the BGW [18] protocol.* As a side contribution we also perform the first computer-aided verification of a variant of the fundamental BGW protocol for passive and static active adversaries (with computational security in the latter case). In what follows, we outline the main contributions of this work.

***Specification and computer-aided proofs of (P)MPC in* EasyCrypt** Our proof is performed in the computation model, using the game-based infrastructure provided by EasyCrypt's engine. We make use of the real/ideal paradigm to specify our security notions. We define an environment that is able to select inputs for each party involved in the protocol. The environment has the ability to trigger an adversary that actively corrupts parties (change their inputs, remove them from the execution, etc.) via oracles. This adversary interacts with an *evaluator* that either retrieves to the adversary information from a real execution of the protocol or from an ideal and simulated one. The adversary redirects this information to the environment that tries to distinguish between the two possibilities mentioned. The malicious setting requires a computational bound in our proofs due to the use of the Decision Diffie-Hellman (DDH) assumption in the underlying verifiable secret sharing (VSS).

To accomplish the computer-aided verification we start by identifying appropriate levels of abstraction for MPC protocols (in terms of specification, security and composition) to allow the simplification and reuse of

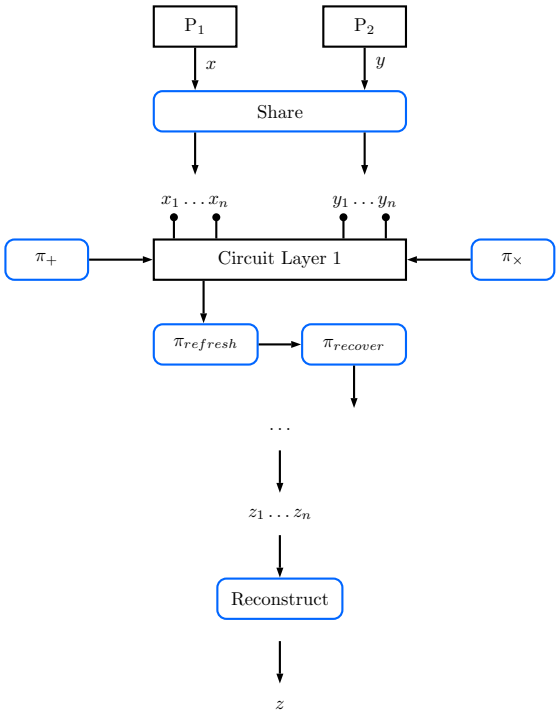

Fig. 1: Overview of the verified (P)MPC evaluator; verified sub-protocols are highlighted in blue. Parties first share their private inputs (via the `Share` protocol) which are then passed to the evaluator. The evaluator interprets computation as an arithmetic circuit using verified implementations of addition (π_+) and multiplication (π_x) protocols based on the BGW [18] protocol. Parties locally compute additions and halt on every multiplication, where they synchronize executions. These protocols can be sequentially composed with the `Refresh` ($\pi_{refresh}$) and `Recover` protocols ($\pi_{recover}$) to provide proactive security. At the end of the evaluation, the result of the computation is obtained by reconstructing the output shares via a `Reconstruct` protocol.

proof steps across the main proof. This abstract structure is an interesting side contribution of this work. It is general enough to be reused since it accommodates many possible instantiations of both MPC and secret sharing protocols. It can be a starting point for other machined-checked MPC proofs to be performed in the future. We also provide a collection of useful lemmas proven in the abstract (such as composition lemmas); this means that any user that wants to leverage advantage of our abstract architecture already has a set of lemmas that can be carried out to concrete instantiations with very little effort and in a mechanical way. Our abstract structure can then be used in other MPC proofs and/or can be tested with different concrete implementations of MPC protocols, other than the ones we provide. We succinctly show how the abstract framework can be reused in Section 3. The proof is completed by providing concrete instantiations that match the abstract definitions, e.g., a variant of the BGW [18] protocol in case of MPC, and in addition to standard and proactive secret sharing, proactive secret sharing for dishonest majorities [32,35,12].

Verified extraction tool-chain In order to obtain a verified implementation of the concrete protocol instantiations defined in EasyCrypt, we developed a new extraction tool-chain for EasyCrypt. We use the Why3 framework [39,38] as an intermediate layer in the extraction process; this allows us to use Why3's new powerful extraction mechanism [60] to obtain verified implementations of EasyCrypt descriptions in multiple target languages. In this work we only extract executable OCaml code, but our pipeline can be extended to extract C code with some additional work.

As a side, an important feature of our verified MPC evaluator, or the (proactive) secret sharing components thereof, is the possibility of applying it in other contexts and in other projects. For example, it can be composed with a verified arithmetic circuit generator to build a verified MPC stack similarly to that for the two-party case in previous work [4].

A Note on Universal Compostability (UC) Our EasyCrypt formalizations are not in the UC framework. In parallel to our work, Stoughton *et al.* formalized in EasyCrypt a UC proof of security of secure message communication using a one-time pad generated using the Diffie-Hellman key exchange [12]. To the best of our knowledge their work does not cover (P)MPC. Extending our work to the UC framework (possibly using the framework of Stoughton *et al.*) is an interesting avenue for future work but outside the scope of this paper.

1.3 Overview of Operation of the Verified (P)MPC Evaluator

The architecture of our evaluator is shown in Figure 1, where verified sub-protocols are highlighted in blue. Computing parties start by sharing their private inputs (via the `Share` protocol), shares are then passed to our evaluator which is running on each party. The evaluator is able to interpret arithmetic circuits using verified implementations of an addition (`add` or π_+) and multiplication (`mul` or π_x) protocols based on a variant of the BGW protocol; in this variant of BGW we use a computationally secure verified secret sharing, VSS, scheme. Parties can locally compute additions and will halt on every multiplication protocol, where they synchronize executions. These protocols can then be combined with the `refresh` (or $\pi_{refresh}$) and `recover` protocols (or $\pi_{recover}$) to achieve proactive security. At the end of the evaluation, the result of the computation can be obtained by reconstructing the output shares of the verified secure evaluator via the `Reconstruct` protocol.

The `Share`, `Reconstruct` and the subsequent protocols for evaluating arithmetic gates are implemented and machine-checked in EasyCrypt. Our specifications and proofs for proactive secret sharing are based on the work by Dolev *et al.* [32], Eldefrawy *et al.* [35] and Baron *et al.* [12] for the dishonest majority setting. When we extend the protocols to MPC, we follow the BGW [18] protocol which only deals with honest majority. One of the important contributions of our work is completing the first computer-aided verification of a variant of the BGW protocol for passive and (static) active adversaries.

Limitations Formalization of underlying mathematical structures and basic components (i.e., finite fields and cyclic groups and randomness generation), as well as the formalization of a Reed-Solomon decoder (e.g., the Berlekamp-Welch algorithm), is out of scope of this work and is left abstract in our EasyCrypt code. Our Trusted Computing Base (TCB) includes precisely the tools used to instantiate these abstract components: Cryptokit, used to instantiate randomness generation, the OCaml zarith library, used to instantiate finite fields and cyclic groups and finally ocaml-reed-solomon-erasure, an OCaml implementation of a Reed-Solomon decoder.

1.4 Paper Outline

In Section 2, we provide informal descriptions of the cryptographic primitives and protocols that are used in this paper. Section 3 describes how EasyCrypt is used to obtain a concrete proof of security for the aforementioned primitives and protocols. We describe in Section 4 our EasyCrypt extraction tool-chain and how it is used to synthesize a concrete implementation of the evaluator. Performance of the extracted implementations is in Section 5. We finish up by summarizing related work in Section 6, and concluding the paper with a discussion of future research directions in Section 7.

[12] `https://github.com/easyuc/EasyUC`

2 Preliminaries: (Proactive) Secret Sharing and MPC

Secret sharing In secret sharing [66,21], a secret s is shared among n parties such that an adversary corrupting up to t parties cannot learn s, while any $t+1$ parties can recover s. A secret sharing protocol consists of two sub-protocols: `Share` and `Reconstruct`. Initially, secret sharing schemes only considered (exclusively) *passive* or *active adversaries*, and later work [52] generalized this to mixed adversaries.

Proactive secret sharing (PSS) The security of secret sharing should be guaranteed throughout the entire lifetime of the secret. The notion of proactive security was first suggested by Ostrovsky and Yung [59], and applied to secret sharing in [50]. It protects against a mobile adversary that can change the subset of corrupted parties over time. Such an adversary could *eventually gain control of all parties* over a long enough period, but is limited to corrupting less than t parties during the same period. In this work, we use the definition of PSS from [52,33]: in addition to `Share` and `Reconstruct`, a PSS scheme contains a `Refresh` and a `Recover` sub-protocols. `Refresh` produces new shares of s from an initial set of shares. An adversary that controls a subset of the parties before the refresh and the remaining subset of parties after, will not be able to reconstruct the value of s. `Recover` is required when one of the participant is rebooted to a clean initial state. In this case, the `Recover` protocol is executed by all other parties to provide shares to the rebooted party. Ideally such rebooting is performed sequentially for randomly chosen parties at a predetermined rate – hence the "proactive" security idea. In addition, `Recover` could be executed after an active corruption is detected. While most of the literature on PSS focuses on the honest majority setting [59,50,71,73,25,11,13,65], the first PSS with dishonest majority was proposed by Dolev et al. in [33]. Standard (linear) secret sharing schemes store the secret in the constant coefficient of a polynomial of degree $< n/2$, an adversary that compromises a majority of the parties would obtain enough shares to reconstruct the polynomial and recover the secret. Instead, [33] leverages the gradual secret sharing scheme of [52] constructed against mixed (passive and active) adversaries, and introduces a PSS scheme robust and secure against $t < n-2$ passive adversaries, or secure but not robust (with identifiable aborts) against either $t < n/2-1$ active adversaries or mixed adversaries (k active corruptions out of $n-k-1$ total corruptions). *As part of our work we implement and verify two versions of PSS. The first PSS is based on the standard Shamir scheme for honest majorities and serves as a foundations for our BGW-based MPC. The second PSS is one for dishonest majority based on gradual secret sharing [33]. However, we stress that we do not have an MPC evaluator for a dishonest majority based on the latter secret sharing scheme.*

Secure Multiparty Computation (MPC) The BGW protocol [18] is one of the first MPC protocols and can be used to evaluate an arithmetic circuit (over a field $\mathbb{F}$) consisting of addition and multiplication gates. The protocol is mainly based on Shamir's secret sharing, where parties share their inputs. Addition can be performed locally, with each party adding its shares bu multiplication requires communication. To evaluate multiplication, parties locally multiply the shares they hold to obtain a $2t$ sharing of the product, then perform a degree reduction by using Lagrange coefficients (this only requires linear operations) to obtain the sharing of the free term of that polynomial.

Proactive MPC (PMPC) PMPC can be regarded as special case of MPC that remains secure in the proactive security model [59]. In [11], Baron et al. constructed the first (asymptotically) efficient PMPC protocol by using an efficient PSS as a building block. The key idea consists in dividing the protocol execution into a succession of two kinds of phases: *operation phases* (the computation of a circuit's layer with addition and multiplication gates, e.g., as in BGW) and *refresh phases*.

3 Machine-checked security proofs for Secret Sharing & MPC

In this section, we first describe what we prove in EasyCrypt, which represents the first step in achieving a verified implementation of an MPC evaluator. Our proof approach consists of two main steps: 1) formalizing in EasyCrypt an abstract framework that captures the desired behavior that we aim for our evaluator; and

2) an instantiation step that provides concrete realizations (also in EasyCrypt) of the protocols and primitives defined in the abstract setting. The intuition is that we will use the abstract framework to perform proofs at a high and modular level, and then propagate those results downwards to concrete implementations with little (proof) overhead. In fact, our abstract framework is modular enough to be reused and applied to other MPC instantiations. We provide a concrete example of this later (see subsection 3.4).

We start the presentation of our EasyCrypt implementation with the description of the abstract framework ($\sim$ 2K LoC), that can be subdivided into two independent modules, one for secret sharing and one for MPC protocols. We want to establish a relation between both in order to reduce the security of the overall evaluator to the security of an arbitrary secret sharing scheme and to the security of an arbitrary PMPC protocol, as shown in Theorem 1. We end the section with an explanation of the instantiation step, accounting for 7K LoC, of which 1K comprises protocol specifications.

3.1 Proof overview

We first give an overview of the proving process, giving a high-level outline of it. As already mentioned, we first modeled the desired behaviour of our evaluator by means of an abstract and modular setting, encompassing the abstract framework to specify abstract functional definitions and security properties for all cryptographic constructions (primitives, algorithms, and protocols) that are used in this work. Since it does not take into account any functional specification of the components involved, it makes sense to use it to establish as much results as possible. Concretely, many results (such as composition theorems) can be obtained by reasoning at an abstract level, which can then be easily carried out to concrete instantiations. For example, one can prove that two primitives (that match some security definitions) can be composed to yield another primitive with a powerful security definition. The construction of a CCA-secure encryption scheme via a CPA-secure encryption scheme and an UF-secure MAC scheme, without specifying any of the two, is a good example of this approach [54].

Our abstract framework is comprised of two abstract modules: one for secret sharing and another for MPC protocols. The first one defines an abstract construction of a secret sharing scheme and specifies three security definitions for it: passive (honest-but-curious), *integrity*, and malicious (verifiable) secret sharing. At this level, we prove that security of a verifiable secret sharing scheme can be reduced to security of a passive secret sharing scheme that ensures integrity of the shares (i.e., any modification of the shares is detectable by the parties involved in the protocol). We also prove that share integrity can be ensured if a commitment scheme is used along side a secret sharing scheme, thus proving that an honest-but-curious secret sharing scheme can be composed with a commitment scheme in order to build a verifiable secret sharing scheme.

The second one provides an abstract formulation of MPC protocols and four security definitions: passive, malicious, *random*, and *proactive*. The two latter security definitions are enough to accommodate the desired behavior of the refresh and recover protocols of the proactive model, respectively. We define a series of (sequential) composition lemmas involving the security notions above, from which we highlight three:

(1) malicious $\circ$ malicious $\Rightarrow$ *malicious*

(2) *random* $\circ$ malicious $\Rightarrow$ *random*

(3) *proactive* $\circ$ *random* $\Rightarrow$*proactive*.

Armed with these three lemmas, we can state that one is able to achieve a *proactive* secure MPC protocol by combining smaller *malicious* MPC protocols with a *random* and a *proactive* protocol. Concretely to our case, we want to have a sequence of add and mul protocols (that perform the evaluation of the desired arithmetic circuit) ending with a refresh and a recover protocol to wrap the security of the execution.

Finally, we use these two abstract constructions to specify Theorem 1 that upper bounds the (proactive) security of the evaluator with the security of the cryptographic constructions it encompasses without considering concrete realizations.

Theorem 1. *For all* ProAct *adversaries* $\mathcal{A}$ *against the* EasyCrypt *implementation* Evaluator*, there exist efficient simulator* S *and adversaries* $\mathcal{B}_{\mathsf{VSS}}$ *and* $B_{\mathsf{Proactive-MalSS}}$*, such that:*

$$\mathsf{Adv}^{\mathsf{Proactive}}_{\mathsf{Evaluator,S}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{VSS}}(\mathcal{B}_{\mathsf{VSS}}) + \mathsf{Adv}^{\mathsf{Proactive-MPC}}(\mathcal{B}_{\mathsf{Proactive-MPC}})$$

where $\mathsf{Adv}^{\mathsf{VSS}}(\mathcal{B}_{\mathsf{VSS}})$ *and* $\mathsf{Adv}^{\mathsf{Proactive-MPC}}(\mathcal{B}_{\mathsf{Proactive-MPC}})$ *represent the advantages against the (verifiable) secret sharing scheme and against the proactive MPC protocol to be executed.*

Our EasyCrypt proof ends with the instantiation step, where concrete realizations of a secret sharing scheme, commitment scheme, and MPC sub-protocols are specified and tied to the abstract definitions. The main challenge in the instantiation step is to prove that the concrete definitions are actually valid instantiations of the desired primitives, since all results obtained in the abstract framework (namely Theorem 1), are easily carried out to the concrete constructions once they are proven to be valid instantiations of primitives.

An important component of this work was the development of a verified EasyCrypt polynomial library, that provides a series of types and operators that deal with polynomials defined over a finite field. We include a description of such library in Appendix Appendix A.

In what remains of this section, we will detail our EasyCrypt implementation, explaining the intermediate steps that were followed in order to derive Theorem 1, namely how to build the two cryptographic primitives that define the security of the evaluator. We will also provide EasyCrypt code snippets for relevant components of this work. Due to space constrains, we will not be able to provide an extensive explanation of the proof, as we will focus on just concrete instantiations of secret sharing schemes or MPC protocols. We refer the reader to the full version of this paper [36] for a more detailed and complete description of our EasyCrypt deployment.

3.2 Secret Sharing in EasyCrypt

Secret sharing, and its verifiable and proactive variations, plays a central role in our (proactive) secure evaluator. It is the first primitive to be executed and any security violation in this component can compromise the entire security of the evaluator. Different shares of different secrets must be indistinguishable and should also be non-corruptible. Our first goal is to formally represent a verifiable secret sharing scheme and to derive a concrete EasyCrypt implementation of it that could be extracted to executable code via our extraction tool-chain.

We first describe our abstract secret sharing framework, pointing out how it can simplify the proofs that could be much more complex if they were to be carried out considering concrete realizations of the involved primitives. We then show how the abstract secret sharing primitives can be instantiated, and how the results proven in the abstract setting are naturally propagated (down) to concrete instantiations of these schemes.

Abstract Secret Sharing The structure of the abstract secret sharing framework is shown in Figure 2. In this figure, we depict abstract cryptographic primitives as rounded rectangles and security definitions as red rectangles. If some security notion applies to some primitive, then the two are connected by a solid arrow. The same way, if a primitive is used as an abstract building block in the security of another primitive, it will be enclosed inside its rounded rectangle as a green circle. Security proofs are represented as dashed arrows. Labeled arrows represent proofs that are propagated to the concrete instantiation of the abstract modular framework.

The main result shown in Figure 2 is the definition of a verifiable secret sharing scheme as the composition of an honest-but-curious secret sharing scheme with an unforgeable commitment scheme. Our formalization of such results starts with the abstract definition of a secret sharing scheme structure, illustrated in Figure 3. A secret sharing scheme is parameterized by 4 integers: i. n - number of parties involved in the protocol; ii. k - number of secrets to be shared; iii. d - number of parties needed to reconstruct the secret; and iv. t

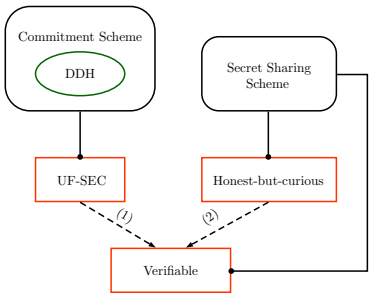

Fig. 2: Secret sharing abstract framework. We define two cryptographic primitives and provide three security definitions, one for commitment schemes and two for secret sharing schemes. The same secret sharing abstraction can be used for both security definitions.

- corruption threshold. The addition of the k parameter may be an exaggeration for most secret sharing protocols but it allows the specification of *batch* secret sharing schemes [42].

Next, the interface defines both the types and operations involved in the sharing protocol, including party identifiers, secrets, shares and randomness types and, finally, the actual share and reconstruct operators. We highlight the inclusion of *p_id_set*, which will contain all party identifiers. This parameter will be very useful when proving correctness properties of both secret sharing and subsequent MPC protocols.

```
theory SecretSharingScheme.
  op n : int. (* Number of parties *)
  op k : int. (* Number of secrets *)
  op d : int. (* Number of parties needed to reconstruct *)
  op t : int. (* Corruption threshold *)
  type p_id_t. (* Party identifier *)
  op p_id_set : p_id_t list. (* Set of parties involved *)
  type secret_t. (* Secret type *)
  type share_t. (* Share type *)
  type shares_t = (p_id_t * share_t) list. (* Set of shares *)
  op dshare : share_t distr. (* Share distribution *)
  type rand_t. (* Randomness *)
  (* Share operation *)
  op share : rand_t -> secret_t -> shares_t.
  (* Reconstruct operation *)
  op reconstruct : shares_t -> secret_t option.
end SecretSharingScheme.
```

Fig. 3: Abstract secret sharing scheme

Randomness is modeled as an explicit operator parameter. This allows us to write probabilistic operations with deterministic flavour. For example, we define the share operator as **op** share : rand_t → secret_t → shares_t, which is a deterministic operator. Nevertheless, the sharing procedure of a secret sharing scheme is, naturally, a probabilistic algorithm. Lifting it to a probabilistic operator would involve writing it as **op** share : secret_t → shares_t distr. Semantically, it means that the operator would be outputting probability distribution on shares and that one would need to sample from this distribution in order to get a valid set of shares. This lift would have very little effect on the proof but would, however, impact the code extraction process as it would be infeasible to extract such probabilistic operators.

We define three security requirements for secret sharing: *honest-but-curious*, *integrity* and *verifiable*. For a full description of these security definitions, we refer the reader to Appendix B for their details.

Abstract commitment scheme To guarantee integrity of generated shares, a dealer needs to commit to them. A commitment scheme can be defined based on the type of randomness used, the type of messages and the type of the commits it produces. Figure 4 shows the abstraction of a commitment scheme. Interestingly, the same abstraction could also be used to define other integrity mechanisms, e.g., message authentication codes.

```
theory CommitmentScheme.
  type rand_t. (* Randomness *)
  type msg_t. (* Message type *)
  type commit_t. (* Commit type *)
  (* Mac operation *)
  op commit : rand_t -> msg_t -> commit_t.
  (* Verify operation *)
  op verify : msg_t -> commit_t -> bool.
end CommitmentScheme.
```

Fig. 4: Abstract commitment scheme

Nonetheless, such abstraction is not enough for the construction of a verifiable secret sharing scheme. In fact, the way the commitment scheme is written, it only works for a single share, instead of multiple shares as desired in our composition proof. We thus define a list variation of a commitment scheme that applies the commitment scheme to a list of messages in order to smooth the composition with an honest-but-curious secret sharing scheme. With this purpose, we defined a new theory ListCommitmentScheme (Figure 5) which is parameterized by any commitment scheme. This means that, in order to come up with a list variation of a given commitment scheme, one only needs to plug the *single* version of the commitment scheme to theory ListCommitmentScheme. The modularity of the secret sharing abstract framework can also be verified here, as the described list version will work for any commitment scheme that is plugged to it.

```
theory ListCommitmentScheme.
  (* Abstract commitment scheme *)
  clone import CommitmentScheme as CS.
  clone import CommitmentScheme as ListCS with
    type rand_t = CS.rand_t,
    type msg_t = CS.msg_t list,
    type commit_t = CS.commit_t,
    op commit (r : rand_t) (m : msg_t) =
      map (CS.commit r) m,
    op verify (m : msg_t) (c : commit_t) =
      all ((=) true) (map2 CS.verify m c).
end ListCommitmentScheme.
```

Fig. 5: Abstract *list* commitment scheme

We specify an unforgeability security definition for commitment schemes that can be consulted in Figure 32 in Appendix B.

```
theory AbstractVerifiableSecretSharing.
  ...
  clone import SecretSharingScheme as AVSS with
    ...
    type share_t = SS.share_t * CS.commit_t,
    type rand_t = SS.rand_t * CS.rand_t,
    op share (r : rand_t) (s : secret_t) =
      let (rs,rc) = r in
      let ss = SS.share rs s in
      let cs = LCS.commit rc ss in
      merge ss cs,
    op reconstruct (css : shares_t) =
      let ss = unzip1_assoc css in
      let cs = unzip2_assoc css in
      if LCS.verify ss cs then SS.reconstruct ss else None.
  ...
end AbstractVerifiableSecretSharing.
```

Fig. 6: Abstract verifiable secret sharing scheme in EasyCrypt.

Abstract verifiable secret sharing We conclude our description of the abstract secret sharing framework by describing our modular composition proof of a semi-honest secret sharing scheme with an unforgeable commitment scheme, yielding a verifiable secret sharing scheme. We do not claim any novelty here, this is a standard technique in previous verifiable secret sharing articles [1,26,37,40,43,44,51,53,55,61,62,63,8,27].

The primitives can be composed as shown in Figure 6. Our composition proof proceeds similar to the *encrypt-then-MAC* for CCA-secure encryption schemes. We leave the passive secret sharing scheme abstract but we force some types of the commitment scheme, namely that it produces commits to associations between a party identifier and its respective share. Thereby, the list variation of this commitment scheme would be producing commits to a map between party identifiers and shares, which is precisely the type of values outputted by algorithm share presented in Figure 3.

The composition of the two primitives is a new secret sharing scheme where shares now carry a commitment with them. Protocols share and reconstruct are specified in the expected way: i. share first shares the secret according to the honest-but-curious secret sharing scheme, before producing commits to the generated shares; and ii. reconstruct first verifies the integrity of the shares (if there was some malicious modifications). If no integrity breach was found, it proceeds with the reconstruction of the secret according to the passive secret sharing scheme. If it was unable to attest the validity of some share, the algorithm fails and produces no output.

Theory AbstractVerifiableSecretSharing also includes a series of security instantiations. The goal is to prove that malicious security for AVSS can be obtained as a combination of other security guarantees, namely the unforgeability of the commitment scheme and the indistinguishability of the passive secret sharing scheme.

The security proof of the verifiable secret sharing construction in Figure 6 was done in two steps. We start by showing that malicious security of AVSS can be reduced to its passive security and its integrity. Finally, we individually show that passive security of the verifiable secret sharing scheme can be reduced to the passive security of the underlying secret sharing scheme and that share integrity can be reduced to the unforgeability of the underlying commitment scheme.

Concrete secret sharing We now show how to make use of the modular abstract framework to produce concrete instantiations of cryptographic primitives that are compatible with our EasyCrypt extraction tool, focusing on the achievement of a verifiable secret sharing scheme. Based on the afore described modular proof, one only needs to provide a valid realization of an honest-but-curious secret sharing scheme and of an unforgeable commitment scheme. The verifiable secret sharing scheme is derived easily via specifications in

Figure 6, as presented in Figure 7. In the figure, if some arrow is labeled, it means that the result is already proven in the abstract framework and that it can be reused here (hence it does not represent any significant verification overhead). The proofs that were made in this phase (proofs that concrete implementation of a primitive achieves the desired security) are represented as solid arrows.

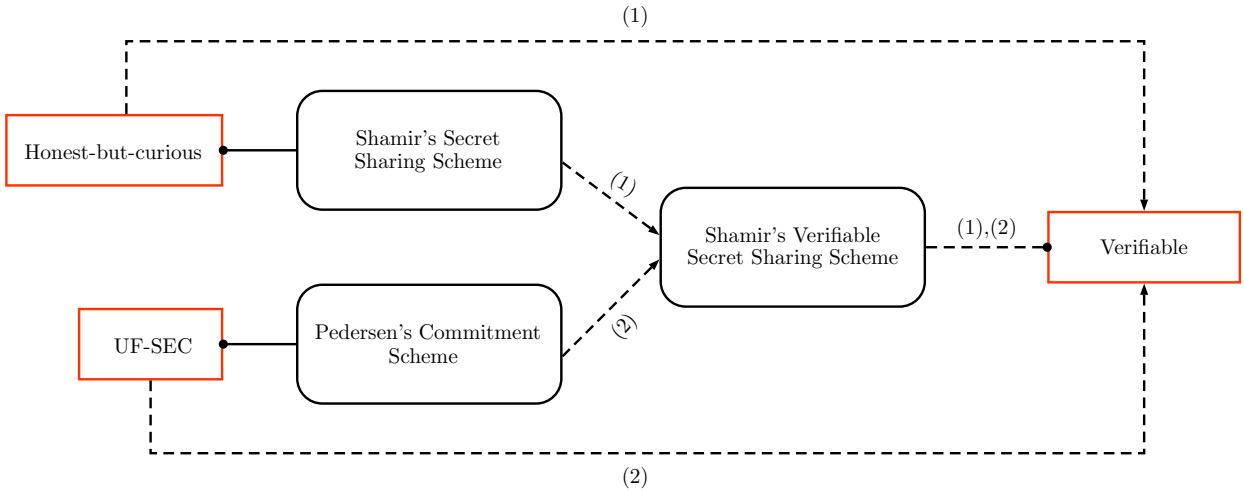

Fig. 7: Concrete instantiation of the secret sharing framework. Concrete Shamir's verifiable secret sharing scheme is easily obtained via a concrete instantiation of Shamir secret sharing scheme and of Pedersen's commitment scheme. The major challenges in this step are the security proofs of Shamir secret sharing schem and of Pedersen's commitment scheme.

For our proactive secure evaluator, we choose to implement the verifiable version of Shamir's secret sharing scheme [67] combined with Pedersen's commitment scheme [62]. The EasyCrypt implementation of Shamir's secret sharing scheme is detailed in Figure 8, where $F.t$ represents the type of elements of a finite field.

```
theory ShamirSecretSharingScheme.
  ...
  clone import SecretSharingScheme as Shamir with
    type secret_t = F.t,
    type share_t = F.t,
    type rand_t = polynomial,
    op share (r : rand_t) (s : secret_t) =
      map (fun x => (x,eval x r)) p_id_set.
    op reconstruct (ss : (p_id_t * share_t) list) =
      Some (interpolate CyclicGroup.FD.F.zero ss).
  ...
end ShamirSecretSharingScheme.
```

Fig. 8: Shamir secret sharing scheme

The next step was to specify a concrete commitment scheme. Because we were interested in exploring the homomorphic properties of commitments, we chose to implement Pedersen's commitment scheme, reproduced in Figure 9.

```
theory PedersenCommitmentScheme.
  ...
  op multiplicator (l : group list) : group =
    foldr (CyclicGroup.( * )) g1 l.
  clone import CommitmentScheme as PedersenCS with
    type rand_t = polynomial * polynomial,
    type msg_t = t * t,
    type commit_t = t * ((int * group) list),
    op commit (r : rand_t) (m : msg_t) =
      let (rs,rc) = r in let (m,sh) = m in
      let rz = zip rs rc in
      (eval m r, map (fun x => ((fst x).`expo, g^(fst x).`coef * h^(snd x).`coef)) rz),
    op verify (m : msg_t) (c : commit_t) : bool =
      let (c, gsh) = c in
      let (m,sh) = m in
      g^sh*h^c = multiplicator (map (fun x => snd x ^ (m ^ fst x)) gsh).
  ...
end PedersenCommitmentScheme.
```

Fig. 9: Pedersen commitment scheme

Finally, we can instantiate the abstract verifiable secret sharing construction using the concrete passively secure secret sharing scheme (ShamirSS, inside theory ShamirSecretSharingScheme) and the concrete unforgeable commitment scheme (PedersenCS, inside theory PedersenCommitmentScheme). This instantiation step is actually very simple and can be done with very little overhead. An EasyCrypt implementation of the verifiable secret sharing version of Shamir's secret sharing scheme can then be obtained simply by plugging ShamirSS and PedersenCS to the abstract verifiable secret sharing construction.

PSS with dishonest majority We also implemented a proactive (gradual) secret sharing scheme [32,35] which is secure against a dishonest majority of mixed adversaries - detailed in Section 4 - where we use it for an example-driven presentation of how the EasyCrypt extraction tool works. Formalization and extracted executables of this scheme can be of independent interest beside the BGW-based MPC part.

3.3 MPC Protocols in EasyCrypt

We divide the MPC sub-protocols we specified into two main categories: 1. *arithmetic protocols*, used to perform actual circuit computation; these are the addition (add) and multiplication (mul) protocols; and 2. *"security" protocols*, used to ensure proactive security throughout the evaluation of the circuit; protocols refresh and recover will be used in this context.

The circuit to be executed will be composed of a series of addition and multiplication protocols, interpolating with refresh and recover in order to maintain the evaluation secure. Informally, we want protocols add and mul to achieve *privacy*, meaning that we will only want for the communication traces and output shares of these protocols to leak no information about the input shares involved in the protocol. Afterwards, an execution of refresh would re-randomize the outputs. We call such security notion *random*. Lastly, recover can be used in order to prevent an adversary from potentially corrupting all nodes of the system. We say protocol recover achieves *proactive* security. All the afore mentioned security notions consider malicious adversaries. This high-level description is summarized in Figure 10, where we depict our abstract modular framework for MPC protocols. This architecture leverages the security definitions found in other simular tools such as Sharemind [22,23] or [5]. Nevertheless, these two works only consider passive security, while our evaluator provides proactive security.

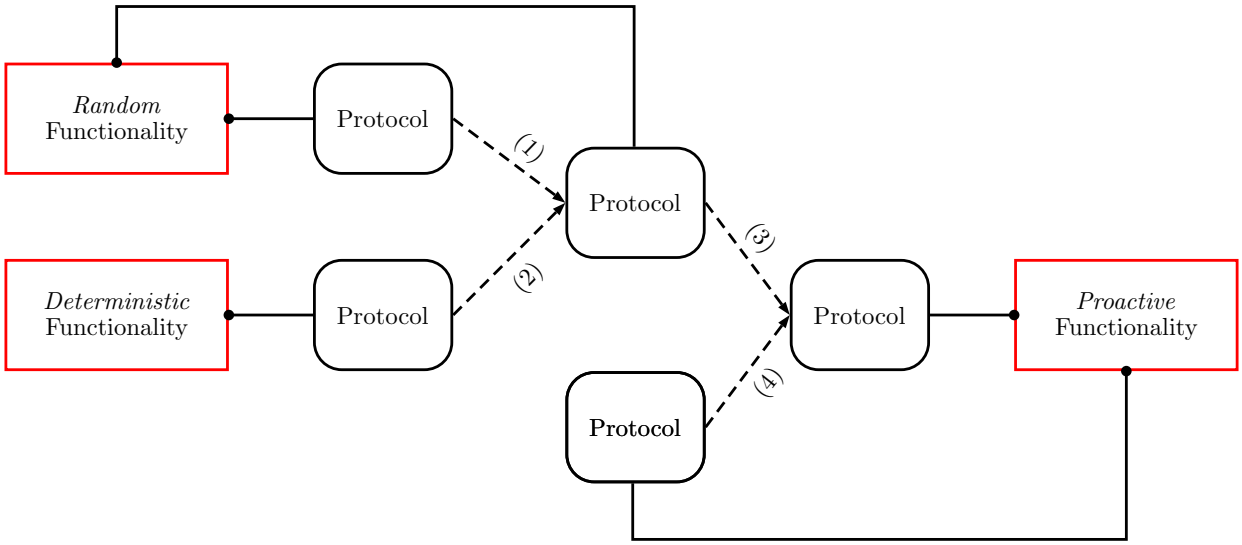

Fig. 10: MPC abstract framework. Protocols can realize three functionalities - *deterministic*, *random* and *proactive*. *Smaller* protocols can also be composed in order to achieve stronger security guarantees.

Abstract MPC At the base of our MPC abstract framework lies the abstraction of an MPC protocol (Figure 11). The abstraction defines an operator *prot* which models the functional behaviour of the protocol, with parties entering the protocol with their inputs and randomness information and ending with some output. In a possible instantiation, one could specify individual party operations that can then be called inside *prot* operator, which would model communication.

```
theory AbstractProtocol.
  type p_id_t. (* Party identifier *)
  op p_id_set : p_id_t list. (* Set of parties involved *)
  type input_t. (* Party input *)
  type inputs_t = (p_id_t * input_t option) list. (* Set of party inputs *)
  type output_t. (* Party output *)
  type outputs_t = (p_id_t * output_t option) list. (* Set of party outputs *)
  type rand_t. (* Party randomness *)
  type rands_t = (p_id_t * rand_t option) list. (* Set of party randomness *)
  type conv_t. (* Party "conversation" *)
  type convs_t = (p_id_t * conv_t option) list. (* Set of party "conversations" *)
  (* Executes the protocol *)
  op prot : rands_t -> inputs_t -> (convs_t * outputs_t).
end AbstractProtocol.
```

Fig. 11: Abstract MPC protocol

The proposed abstraction not only fits complete instantiations of protocols but also allows one to *split* the same protocol into multiple *smaller* protocols and then compose them into a *bigger* protocol. For example, the refresh protocol can be written as a composition of a protocol where every party shares the value 0 (zero), a protocol where every party sends to each other the corresponding shares and a protocol where every party adds the received shares to the current share. This approach is actually ideal when reasoning

about the security of a protocol against malicious adversaries. Intuitively, one can prove the desirable level of security for the individual protocols and, subsequently, derive the security of the composed protocol using the composition lemmas that we describe with more detail later in this section. Note that this approach contemplates possible misbehaviors in the middle of the protocol execution, which is a desirable property for malicious secure protocols and, probably, the most challenging aspect to model.

In our evaluator, MPC protocols may realize three types of functionalities. Similarly to the protocol abstraction, we define functionalities that will execute in one block and functionalities that will execute by phases. In the *private* functionality, protocols must realize some arithmetic operation. Intuitively, this functionality receives unshared inputs and computes the arithmetic operation over raw inputs instead of over shared values. Regarding *random* functionality, protocola must re-randomize the input shares. In short, executing the protocol must output a result indistinguishable from freshly sharing a secret after reconstructing it. Finally, in the *proactive* functionality, protocols must recover a corrupted party to a non-corrupted state. Informally, recovering parties should receive new shares that would invalidate the previous ones. EasyCrypt snippets for the three functionalities can be found in the extended version of this paper.

These functionalities are tied to a specific $Real \sim Ideal$ security experiences. For *private* and *random* security, our security experiencies are extensions to the malicious setting of the ones defined in [22,5], while for *proactive* security we use the ones formally defined in [35]. Likewise, we define three different types of simulators. This is due to the fact that simulators will have different goals, according to the security asset.

We modeled the security definitions by means of EasyCrypt modules in the following way. An *environment* Z will try to distinguish between a real execution of the protocol or the simulated one. In order to do so, Z can ask an *adversary* A to execute either the protocol or to simulate it (behaviour defined by some bit b). Since we are interested in malicious adversaries, this adversary will be grant access to oracles (that are to be executed at the end of every protocol stage):

- *corrupt* - getting access to party
- *corruptInput* - modify some corrupted party phase input
- *abort* - remove some party from the protocol

At the end, A will provide Z with information (depending on the security defintion) that Z will use to make its guess. We refer the reader to Appendix C for a complete view of the security definitions representation.

The functionalities, security experiences and simulators described above define security for individual protocols but not for the entire evaluator. Intuitively, we want the overall evaluator to be a composition of smaller protocols for which it is simpler to prove security. Subsequently, we want to derive security for the entire system by relying on composition lemmas surrounding the MPC protocols. Informally, we want to evaluate a sequence of addition and multiplication protocols, keeping the evaluation *private*. The circuit will then reach a refresh or a recover protocol, and will assume the security each protocol provides. We thus need three composition lemmas:

1. The composition of two *private* protocol yields a *private* protocol.
2. The composition of a *private* protocol with a *random* protocol yields a *random* protocol.
3. The composition of a *random* protocol with a *proactive* protocol yields a *proactive* protocol.

To finish this subsection, we highlight how EasyCrypt can be used in order to define the composition of a *private* and a *random* protocols and how to specify its security.

Figure 12 details the specification of the composed protocol $\Pi = \pi_2 \circ \pi_1$. The structure is simple. Each party i involved in protocol Π will input two shares x_i and y_i of the values x and y, respectively, and will end the execution of Π with one share z_i of some value z, that will be the randomized result of the arithmetic operation realized by protocol π_1. Inputs x_i and y_i are also the inputs of protocol π_1, whereas the the input of π_2 will be the output of π_1.

The security of the composed protocol can now be reduced to the security of the two protocols. The interesting aspect about this proof is that it can be performed in an abstract and modular way. Consequently, in future instantiations, the only concern will be proving security for the small components of a bigger protocol, since the composition lemmas can be applied. The reason why the proof can be modular is because the

composed simulator and functionalities do not need to be restricted to any particular behavior or description but can simply be defined as the sequential execution of S_1 and S_2 and F_1 and F_2. Figure 12 demonstrates the security set up for the composed protocol Π.

The intuition behind the security proof is that every real execution of a protocol can be replaced by the ideal one. At the end, instead of executing the two protocols sequentially, the security experience will be executing the two simulators sequentially, therefore connecting the real and ideal worlds.

```
theory RandomAfterPrivateComposition.
  ...
  clone import AbstractProtocol as P1.
   clone import PrivateProtocolSecurity as P1PrivSec with
    theory AbstractProtocol <- P1.
  clone import AbstractProtocol as P2 with
    type input_t = P1.output_t.
  clone import RandomFunctionality as RandomFunc with
    type input_t = P1PrivSec.F.output_t.
  clone import RandomProtocolSecurity as P2RandSec with
    theory AbstractProtocol <- P2,
    theory F <- RandomFunc.
  clone import AbstractProtocol as P with
    type input_t = P1.input_t,
    type output_t = P2.output_t,
    type rand_t = P1.rand_t * P2.rand_t.
  clone import RandomProtocolSecurity as PRandSec with
    type F.input_t = P1PrivSec.F.input_t,
    type F.output_t = P2RandSec.F.output_t,
    type F.rand_t = P2RandSec.F.rand_t,
    type F.leak_t = P1PrivSec.leak_t * P2RandSec.leak_t,
    op F.func (r : F.rand_t) (xx : F.input_t) =
      let (y, l1) = P1PrivSec.F.func xx in
      let (z, l2) = P2RandSec.F.func r y in (z, (l1, l2)),
    theory AbstractProtocol <- P.
  ...
  module Simulator(S1 : P1PrivSec.Sim_t) (S2 : P2RandSec.Sim_t)={
    proc simm(l : leak_t, xi : input_t, zi : output_t) = {
      ...
      (l1, l2) <- l;
      (yi, v1) <@ S1.simm(l1, xi);
      v2 <@ S2.simm(l2, yi, zi);
      return ((v1,v2));
    }
  }
  ...
end RandomAfterPrivateComposition.
```

Fig. 12: Composition of a *private* protocol with a *random* protocol

Concrete MPC Instantiation We implemented in EasyCrypt protocols add, refresh and recover specified in [35]. Protocols add and refresh do not introduce a significant implementation overhead since add is a simple local protocol and refresh can be seen as a composition of share (of a secret equal to 0) and add. Protocol recover is slightly more complex than the previous two. It involves non-recovering parties to sub-share their shares and

recovering parties performing polynomial interpolation in order to obtain the new shares. For multiplication, we choose to implement the simplified malicious version of the BGW multiplication protocol [17] described by Asharov and Lindell in [7]. This version is similar to the passive version, with the introduction of a an error correction step that can correct possible subshare corruptions. Figure 13 compiles the instantiation step of MPC protocols.

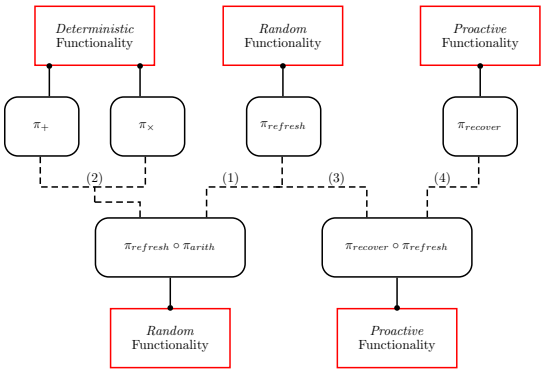

Fig. 13: Concrete instantiation of MPC framework. π_+ and $\pi_\times$ are used to evaluate an arithmetic, while $\pi_{refresh}$ and $\pi_{reconstruct}$ are used to ensure security of the evaluation. Security of the composed protocols is easily derive from the abstract framework once security is proven for concrete instantiations.

We now present the EasyCrypt formalization of the recover protocol. Protocols add, refresh and mul can be found in Appendix D. The EasyCrypt code of the MPC protocols is written party-wise, meaning that we have operators for every stage of the protocol that are then merged in the prot operator. For example, operator nr_pstage1, nr_pstage2 and r_pstage1 in Figure 14 represent the first stage of non-recovering parties, the second stage of non-recovering parties and the first and only stage of recovering parties in the recover protocol, respectively. The outputs of those stages are the party state (values that need to be carried out throughout the execution of the party) and some message, which can either be a broadcast (like bdcst1_t in nr_pstage1) or a simple message (like msg2_t in nr_pstage2). The prot operator is used to depict the global execution of the protocol, including simulation of party communication.

Following the BGW protocol description from [7], we built a Reed-Solomon code specification in EasyCrypt, that we use to correct possible errors in the shares. We keep this definition abstract for now, and leave its concrete realization for future work.

A simple circuit evaluation function can now be defined. This function, taking a circuit description as input (a series of additions and multiplications), outputs a protocol version of that circuit mapping every addition into an add protocol and every multiplication into a mul protocol. Nevertheless, the protocol description of the circuit should also take care of the introduction of periodic calls to refresh and recover protocols, in order to ensure that the system achieves the desired level of security. We opt to introduce a refresh and a recover protocol at the end of every circuit layer. This decision introduces a performance penalty that could be mitigated if recover and refresh would only be introduced in strategic points of the circuit.

3.4 Reusability of the abstract framework

One of the most interesting features of our work is the propose abstract framework. We claim it is general enough to accommodate multiple possible instantiations of both secret sharing schemes and MPC protocols. The previously shown instantiations are just an example of how it can be used. In this section, we demonstrate that our framework is even able to abstract existing MPC platforms such as Sharemind [22,23].

Sharemind incorporates a 3-out-of-3 additive secret sharing scheme, which shares elements of a finite field and produces shares which are also elements of the same finite field. The sharing and reconstructing processes

```
theory RecoverProtocol.
  ...
  clone import Protocol as Recover with
    type pstate1_t = shares_t,
    op nrpstage1 (r : rand_t) : pstate1_t =
      with r = RRP _ => []
      with r = RNRP rs => share rs F.zero,
    op nrpstage2 pid i ss =
      foldr (fun (x : share_t) (acc : share_t) =>
            if verify (pid, fst x) (snd x) then
              AdditionProtocol.pexec (x,acc)
            else acc) i ss,
    op rparty pid r ss =
      with r = RRP rc =>
        let o = interpolate pid (map (fun pidsh => (fst pidsh, fst (snd pidsh))) ss) in
        let c = commit rc (pid,o) in (o,c)
      with r = RNRP _ => (witness, witness),
    op prot rs iss =
      let rp = fst (snd (head witness iss)) in
      let nrp = rem rp p_id_set in
      let iss = map (fun pidi => let (pid,i) = pidi in (pid, snd i)) iss in
      let pst1 = map (fun pid => (pid, nrpstage1 (oget (assoc rs pid)))) nrp in
      let cs = map (fun pid => (pid, map (fun ss => snd ss) (oget (assoc pst1 pid)))) nrp in
      let sss = map (fun pid => (pid, map (fun idss => oget (assoc (snd idss) pid)) pst1)) nrp in
      let os = map (fun pid => (pid, nrpstage2 pid (oget (assoc iss pid)) (oget (assoc sss pid))))
          nrp in
      let cs = map (fun pid => (pid, oget (assoc cs pid) ++ (oget (assoc sss pid)))) nrp in
      let orp = rparty rp (oget (assoc rs rp)) os in
      let crp = map (fun x => snd x) os in
      Some ((rp, crp) :: cs, (rp, orp) :: os).
end RecoverProtocol.
```

Fig. 14: Recover protocol

are conceptually easy. Sharing outputs three shares, where two are randomly generated and the other one is obtaining by subtracting the secret by the two random shares. The secret can be easily reconstructed by adding all the shares. We show how to instantiate this scheme using our secret sharing abstraction in Figure 15.

Similarly to our addition protocol, Sharemind addition (depicted in Figure 16) can be locally computed by simply adding shares.

Refreshing shares inside the Sharemind platform involves parties generating a random value and sending that value to the *next* party, i.e. party i sends its random value to party $i+1$. Shares are randomized by adding the party's random value and subtracting the received random value. Sharemind's refresh protocol can be found in Figure 17.

Finally, the multiplication protocol is slightly more complex than the previous two but is conceptually similar to the refresh protocol as parties will also send information to the party with the next identifier. It involves two calls to the refresh protocol and every party will send the new randomized shares to the *next* party. At the end, the multiplication of two values $\bar{x} = x_1 + x_2 + x_3$ and $\bar{y} = y_1 + y_2 + y_3$ can be obtained by performing $\sum_{i=1}^{3} \sum_{j=1}^{3} x_i \cdot y_j$. We show the EasyCrypt instantiation of Sharemind's multiplication protocol in Figure 18.

```
theory AdditiveSecretSharingScheme.
  clone import SecretSharingScheme with
    op n = 3,
    op k = 1,
    op d = n,
    op t = n - 1,
    type p_id_t = t,
    op p_id_set = [ofint 0, ofint 1, ofint 2],
    type secret_t = t,
    type share_t = t,
    type rand_t = t * t,
    op share (r : rand_t) (s : secret_t) : (int * share_t) list =
      let (r1,r2) = r in
      zip p_id_set [r1; r2; s - r1 - r2],
    op reconstruct (ss : (int * share_t) list) : (secret_t) option =
      Some (summation (unzip2 ss)).
end AdditiveSecretSharingScheme.
```

Fig. 15: Sharemind's additive secret sharing scheme

```
theory SharemindAddition.
  clone import Protocol as SAddition with
    type p_id_t = p_id_t,
    op p_id_set = p_id_set,
    type input_t = share_t * share_t,
    type output_t = share_t,
    type rand_t = unit,
    type conv_t = unit,
    op prot (r : (p_id_t * rand_t) list) (is : (p_id_t * input_t)) =
      (map (fun pid => (pid, None)) p_id_set,
      map (fun pid => let x = oget is.[pid] in (pid, fst x + snd x)) p_id_set).
end AdditiveSecretSharingScheme.
```

Fig. 16: Sharemind's addition protocol

We demonstrated that all Sharemind components are actually concrete instantiations of our abstract framework, thus providing evidence of its modularity and generality. All this elements can then be composed in order to obtain the full Sharemind evaluation system.

4 EasyCrypt extraction tool-chain

Our verified implementation of a (proactive) MPC evaluator is obtained via a new extraction tool-chain for EasyCrypt. The general execution pipeline of the tool-chain is shown in Figure 19. Briefly, an EasyCrypt description is first translated into a WhyML program, which can then be fed to the Why3 [39,38] platform in order to perform extraction to OCaml using the new Why3 (verified) extraction mechanism [60]. Note that besides making use of Why3 code generation capabilities, one could also use Why3's proving system. For example, one could take the generated WhyML program, annotate it with the desired predicates and use Why3 to discharge the generated verification conditions; for example, one can use Why3 to prove safety about some EasyCrypt code in an automated way. This subject is outside the scope of this work and is an interesting future research direction.

```
theory SharemindRefresh.
  clone import Protocol as SRefresh with
    type p_id_t = p_id_t,
    op p_id_set = p_id_set,
    type input_t = share_t,
    type output_t = share_t,
    type rand_t = t,
    type conv_t = t * t,
    op prot (r : (p_id_t * rand_t) list) (is : (p_id_t * input_t)) =
      (map (fun pid => let ri = oget r.[pid] in
                       let ri1 = oget r.[pid - 1] in
                       (pid, (ri, ri1))) p_id_set,
      map (fun pid => let ri = oget r.[pid] in
                      let ri1 = oget r.[pid - 1] in
                      let i = oget is.[pid] in
                      (pid, i + ri - ri1)) p_id_set).
end SharemindRefresh.
```

Fig. 17: Sharemind's refresh protocol

Why3 in a nutshell Why3 is a framework for deductive verification of programs. It allows the user to specify, annotate, prove programs and, if desired, obtain concrete correct-by-construction implementations of the specifications made. Why3 was geared towards automation of proofs by making use of external automatic theorem provers. Nevertheless, it can also be paired with interactive provers such as Coq or Coq.

Why3 incorporates a ML-like language called WhyML. Besides providing features commonly found in other functional programming languages (pattern matching, records, ...), WhyML encompasses an annotation mechanism, allowing an user to write contracts (pre- and post-conditions and loop invariants) for the specified functions. The validity of these contracts can then be checked usingWhy3, that offers a graphical interface to the user to interact with the proving system.

WhyML code may also be used with the objective of generating correct-by-construction executable code. Why3 code extraction mechanism is general enough to support extraction to multiple platforms, such as OCaml or C, by providing the desired driver to the extraction system. In fact, it is also possible to provide user-defined drivers, if one wants to deviate from how Why3 performs extraction by default or if one wants to specify how abstract functions in the WhyML file are to be extracted.

We chose Why3 as an intermediate tool for two reasons. First, the specification languages of both frameworks (EasyCrypt and Why3) are very similar, which simplifies the translation process between the two platforms. Second, Why3 incorporates a powerful, verified, extraction mechanism, supporting extraction to multiple platforms and languages. However, some are not as mature (or verified) as the OCaml extraction. We view our tool-chain as an interesting starting point for a future (more general) EasyCrypt extraction tool-chain that makes use of a refined Why3 based code generation with support for multiple target languages.

From EasyCrypt to OCaml We provide here an example illustration of our tool-chain using gradual secret sharing. We use the (proactive) gradual secret sharing scheme presented in [32,35] to demonstrate how executable code can be obtained from an EasyCrypt protocol specification. Briefly, the gradual secret sharing scheme is a composition of an additive secret sharing scheme and a *batch* secret sharing scheme. The additive secret sharing scheme is first executed using a secret s with the purpose of obtaining d summands $s_1, \ldots, s_d$ adding up to s. Then, every summand is shared linearly, increasing the degree of the sharing polynomial as parties advance on the summand they are sharing. We start by showing how the additive secret sharing scheme was specified in EasyCrypt in Figure 20. Note that Sharemind's additive secret sharing scheme in Figure 15 can be seen as a special instantiation of this one, with the value of n being set to 3.

```
theory SharemindMultiplication.
  clone import Protocol as SMultiplication with
    type p_id_t = p_id_t,
    op p_id_set = p_id_set,
    type input_t = share_t * share_t,
    type output_t = share_t,
    type rand_t = t * t,
    type conv_t = t * t * t * t,
    op prot (r : (p_id_t * rand_t) list) (is : (p_id_t * input_t)) =
      let xx = map (fun pid => (pid, fst (oget is.[pid]))) p_id_set in
      let yy = map (fun pid => (pid, snd (oget is.[pid]))) p_id_set in
      let r1 = map (fun pid => (pid, (oget r.[pid]).`1)) p_id_set in
      let r2 = map (fun pid => (pid, (oget r.[pid]).`2)) p_id_set in

      let (crxx, rxx) = SRefresh.prot r1 xx in
      let (cryy, ryy) = SRefresh.prot r2 yy in

      (map (fun pid => let xi = oget rxx.[pid] in
                       let yi = oget ryy.[pid] in
                       let xi1 = oget rxx.[pid - 1] in
                       let yi1 = oget ryy.[pid - 1] in
                       (pid, (xi, yi, yi1, xi1)) p_id_set,
      map (fun pid => let xi = oget rxx.[pid] in
                      let yi = oget ryy.[pid] in
                      let xi1 = oget rxx.[pid - 1] in
                      let yi1 = oget ryy.[pid - 1] in
                      (pid, xi * yi + xi * yi1 + xi1 * yi)) p_id_set).
end SharemindMultiplication.
```

Fig. 18: Sharemind's multiplication protocol

The additive secret sharing is very easy to implement. Every party will get a random share (given by the random type) except for the first party, who gets the difference between the secret and the summation of all the random summands. The secret can be easily reconstructed by adding all the shares.

The *batch* secret sharing scheme is a good example of a secret sharing scheme that produces shares to mutliple secrets. Briefly, it makes use of one linear secret sharing scheme to share every secret. In order to reconstruct, parties can recover one polynomial at a time and then evaluate it in the zero point to obtain the original secret.

The gradual secret sharing scheme can finally be specified as shown in Figure 22, where we assume that the scheme was instantiated for 15 parties.

We translate every type definition and functional operator to its counterpart in WhyML. We will focus only on the types and operators defined by GradualSS and omit definitions of dependencies such as list operations. The WhyML code generated for the same scheme can be found in Figure 23. As Figures 22 and 23 show, the code in the two scripts is very similar. Abstract values (both abstract operators and abstract constants) are defined in Why3 using `val`, every EasyCrypt **op** is mapped to an WhyML `let` and the *type* keyword is the same in both languages. Note, however, that the order of parametric types is reversed in both languages. In general, our translation tool performs a simple syntactic translation between the two languages, which increases our confidence about correctness of the tool even without a formal proof for this part.

The WhyML program is now ready for extraction. Applying the Why3 extraction mechanism yields the correct-by-construction OCaml code in Figure 24.

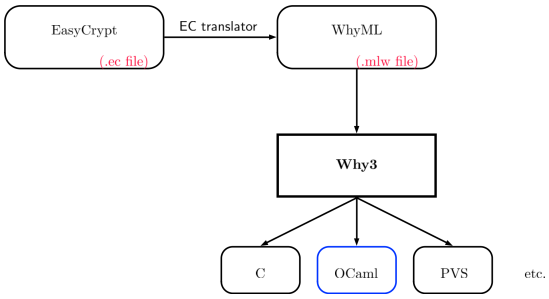

Fig. 19: EasyCrypt extraction tool-chain. We only extract OCaml code in this work, but there are resources online suggesting that C and PVS code could also be extracted from WhyML.

```
theory AdditiveSecretSharingScheme.
  const n : {int | 2 <= n} as gtn_2.
  op k = 1.
  op d = n.
  op t = n - 1.
  type p_id_t = int.
  op p_id_set = iota_ 0 n.
  type secret_t = t.
  type share_t = t.
  type rand_t = t list.
  op share (r : rand_t) (s : secret_t) : (int * share_t) list =
    zip p_id_set ((s - summation r) :: r).
  op reconstruct (ss : (int * share_t) list) : (secret_t) option =
    Some (summation (unzip2 ss)).
  clone import SecretSharingScheme as ASS with
    ...
end AdditiveSecretSharingScheme.
```

Fig. 20: Additive secret sharing scheme

5 Verified (proactive) secret sharing and MPC implementation

Our verified implementation of the secure arithmetic evaluator was obtained via the EasyCrypt extraction tool described in Section 4, except the abstract libraries for randomness generation, Reed-Solomon codes and the cyclic group and finite field structures, which were implemented using the CryptoKit library [13]. We generated cyclic group (and finite field) parameters using openssl and used ocaml-reed-solomon-erasure [14] for instantiating the Reed-Solomon error correction.

In this work we only focus on extracting an OCaml implementation (as highlighted in Figure 19). Because there are no publicly available implementations of proactive secret sharing and PMPC protocols, we compare the execution time of our extracted gradual secret sharing scheme with an unverified manual implementation of gradual secret sharing that we develop using the native secret sharing class in Charm [15]. We provide microbenchmarks for our extracted implementation in both the passive and malicious cases.

Benchmarking results of the secret sharing schemes are presented in Table 3, while benchmarks of the MPC protocols are shown in Table 4. We provide individual performance times for share, reconstruct, add, mult, refresh and recover for a field size of 128, 256, 512 and 1024 bits and for 5, 9 and 15 parties. We present

[13] https://github.com/xavierleroy/cryptokit
[14] https://gitlab.com/darrenldl/ocaml-reed-solomon-erasure
[15] https://github.com/JHUISI/charm

```
theory BatchSecretSharingScheme.
  import PedersenCommitmentScheme.
  const n : { int | 5 <= n < F.q } as ge2n_gtnZPp.
  const d = n - 3.
  const t = n\%/2.
  const k = d.
  type p_id_t = t.
  op p_id_set : t list.
  type secret_t = t list.
  type share_t = (t * commit_t) list.
  type rand_t = polynomial list * polynomial list.
  op exec (p : polynomial) (ss : (p_id_t * share_t) list) (rc : polynomial): (p_id_t * share_t)
      list =
    map (fun id_ss => let (id,ss) = id_ss in (id, ss ++ [(eval id p, commit (p,rc) (id, eval id p))
        ])) ss.
  op loop (rr : (polynomial * polynomial) list) ss =
    with rr = [] => ss
    with rr = ps :: rr' => loop rr' (exec ps.`1 ss ps.`2).
  op share (r : rand_t) (s : secret_t) : (p_id_t * share_t) list =
    let (rs,rc) = r in
    let rr = zip rs rc in
    loop rr (map (fun id => (id, [])) p_id_set).
  op ocons (x : 'a) (ol : 'a list option) =
   omap ((fun x y => x :: y) x) ol.
  op build_points_pid' (pid : p_id_t) (ss : share_t) : ((p_id_t * t) list) option =
    with ss = [] => Some []
    with ss = x :: xs =>
      if verify (pid, x.`1) x.`2 then
        ocons (pid, x.`1) (build_points_pid' pid xs)
      else None.
  op build_points_pid (s : (p_id_t * share_t)) : ((p_id_t * t) list) option =
    build_points_pid' (fst s) (snd s).
  op build_points_pids (ss : (p_id_t * share_t) list) =
    with ss = [] => Some []
    with ss = x :: xs =>
      let ox = build_points_pid x in
      if ox <> None then
        ocons (oget ox) (build_points_pids xs)
      else None.
  op build_points (ss : (p_id_t * share_t) list) =
    let obpt = build_points_pids ss in
    if obpt <> None then
      Some (transpose (oget obpt))
    else None.
  op open_loop (pts : (p_id_t * t) list list) : t list =
    with pts = [] => []
    with pts = x :: xs =>
      interpolate F.zero x :: open_loop xs.
  op reconstruct (ss : (p_id_t * share_t) list) : secret_t option =
    let obpt = build_points ss in
    if obpt <> None then
      Some (open_loop (oget obpt))
    else None.
  clone import SecretSharingScheme as BSS with
    ...
```

Fig. 21: *Batch* secret sharing scheme

```
theory GradualSecretSharingScheme.
  const n = 15.
  op k = BSS.k.
  op d = BSS.d.
  op t = BSS.t.
  type secret_t = ASS.secret_t.
  type share_t = BSS.share_t.
  type rand_t = ASS.rand_t * BSS.rand_t.
  op share (r : rand_t) (s : secret_t) : (BSS.p_id_t * share_t) list =
    let (ra,rb) = r in
    let summands = ASS.share ra s in
    BSS.share rb (map snd summands).
  op reconstruct (ss : (BSS.p_id_t * share_t) list) : secret_t option =
    let summands = BSS.reconstruct ss in
    ASS.reconstruct (zip ASS.p_id_set (oget summands)).
  clone import SecretSharingScheme as GradualSS with
    ...
end GradualSecretSharingScheme.
```

Fig. 22: Gradual secret sharing scheme (File name: *Gradual.ec*).

```
module GradualSS
  ...
  val n : int
  let k = 1
  let d = n - 3
  let t = d - 1
  type p_id_t = int
  val p_id_set : t list
  type secret_t = t
  type share_t = list  (t * (t * (list (int * group))))
  type rand_t = list t * (list (monomial list))
  let share (r : rand_t) (s : secret_t) : list (BSS.p_id_t * share_t)=
    let (ra,rb) = r in
    let summands = share ra s in
    share rb (map snd summands)
  let reconstruct (ss : list (p_id_t * share_t)) : option secret_t =
    let summands = reconstruct ss in
    reconstruct (zip p_id_set1 (oget summands))
```

Fig. 23: Gradual secret sharing scheme in WhyML.

the execution times in milliseconds. We also provide an individual comparison with the Charm unverified implementation of the gradual secret sharing scheme in Table 2.

Our measurements were conducted on an x86-64 Intel Core i5 clocked at 2.4GHz with 256KB L2 cache per core. The extracted code was compiled with ocamlopt version 4.05.0 and the tests were ran in isolation, using the OCamlSys.time operator to read the execution time. We ran tests in batches of 100 runs each, noting the median of the times recorded in the runs.

We found that the sharing operation is faster in our generated code while reconstruct performs better in the Charm version. This may be justified by the verification penalty induced by the verified polynomial library (specially the polynomial interpolation) that only manifests itself in the reconstruction phase. In fact, the share protocol does not make use of significant polynomial operations (besides polynomial evaluation

```
let d1 : Z.t = Z.sub (Z.of_string "5") (Z.of_string "3")
let k : Z.t = d
let t : Z.t = Z.sub d Z.one
type p_id_t = Z.t
type secret_t = Z.t
type share_t = ((Z.t) * ((Z.t) * (Z.t * (Z.t list))) list) list
type rand_t = ((Z.t) list) * ((monomial list) list)
let share2 (r : ((Z.t) list) * ((monomial list) list)) (s : (Z.t)) : ((Z.t) * ((Z.t) * (Z.t * (Z.t
    list))) list) list =
  let (ra,rb) = r in
  let summands = share ra s in
  share1 rb (map snd summands)
let reconstruct2 (ss : ((Z.t) * ((Z.t) * (Z.t * (Z.t list))) list) list) : (Z.t) option =
  let summands = reconstruct ss in
  reconstruct1 (zip p_id_set1 (oget summands))
```

Fig. 24: Gradual secret sharing scheme in OCaml (File name: *gradual.ml*).

Table 2: Performance Benchmark Table Gradual for extract gradual secret sharing and for Charm unverified implementation for a 512 bit-size field

		Share	Reconstruct
Charm	$n = 5$	6	0.3
	$n = 15$	23	2
Extracted	$n = 5$	0.19	0.6
	$n = 5$	1.5	6

that does not represent a major overhead in the execution time) and so its performance depends on the performance of cyclic group and finite field operations. Since these are implemented with the Zarith library [16] (an OCaml wrapper for GMP), we are able to achieve a good performance for it. This is not the case for the reconstruction protocol, whose performance is greatly influenced by the performance of the polynomial library. By relying on a unverified library, Charm delivers better results for polynomial operations, thus delivering faster times for the reconstruct protocol.

Table 3: Performance Benchmark Table for Extracted Secret Sharing Implementation (times in *ms*)

		Shamir				Pedersen	
		Passive		Malicious		Unforgeable	
		Share	Reconstruct	Share	Reconstruct	Commit	Verify
128 bits	$n = 5$	0.0200	0.0523	0.1546	0.0229	0.0566	0.0211
	$n = 9$	0.0458	0.2074	0.4702	0.0332	0.0941	0.0300
	$n = 15$	0.1249	0.5783	1.2102	0.0440	0.1571	0.0497
256 bits	$n = 5$	0.0229	0.0988	0.5746	0.0870	0.2468	0.0618
	$n = 9$	0.0978	0.4433	1.7046	0.1187	0.4410	0.0848
	$n = 15$	0.1853	0.9847	4.8086	0.1814	0.6432	0.1076
512 bits	$n = 5$	0.0290	0.1722	2.2661	0.3532	1.2448	0.2430
	$n = 9$	0.1255	0.7247	7.3145	0.5302	1.7492	0.2347
	$n = 15$	0.2610	1.7041	20.3639	0.7907	2.8111	0.2917
1024 bits	$n = 5$	0.0588	0.3910	15.0708	2.3797	7.1856	1.2635
	$n = 9$	0.1866	1.2866	48.9167	3.6002	12.0770	1.3364
	$n = 15$	0.5989	3.6757	135.1714	5.3811	19.2705	1.5435

[16] https://github.com/ocaml/Zarith

Table 4: Performance Benchmark Table for Extracted (P)MPC Implementation (times in *ms*). First column is the field size, the second is the number of parties.

		Addition				Multiplication				Refresh				Recover			
		Passive		Malicious		Passive		Malicious		Passive		Malicious		Passive		Malicious	
		Total	Per party	Total	Per party	Total	Per party	Total	Per party	Total	Per party	Total	Per party	Total	Per party	Total	Per party
128 bits	$n = 5$	0.0032	0.0006	0.0071	0.0014	0.0790	0.0158	1.6393	0.3279	0.0756	0.0151	1.2306	0.2461	0.1191	0.0238	0.9878	0.1976
	$n = 9$	0.0080	0.0009	0.0133	0.0015	0.4197	0.0466	8.0503	0.8945	0.3971	0.0441	6.1809	0.6868	0.6760	0.0751	5.6529	0.6281
	$n = 15$	0.0161	0.0011	0.0399	0.0027	1.8872	0.1258	35.3512	2.3567	1.8390	0.1226	27.9059	1.8604	2.9788	0.1986	26.7091	1.7806
256 bits	$n = 5$	0.0043	0.0009	0.0095	0.0019	0.1069	0.0214	6.4831	1.2966	0.1005	0.0201	4.6022	0.9204	0.1678	0.0336	3.6051	0.7210
	$n = 9$	0.0067	0.0007	0.0220	0.0024	0.6038	0.0671	33.8347	3.7594	0.5655	0.0628	24.8867	2.7652	1.0586	0.1176	22.1240	2.4582
	$n = 15$	0.0158	0.0011	0.0507	0.0034	2.7882	0.1859	148.1338	9.8756	2.6609	0.1774	110.5301	7.3687	4.6013	0.3068	103.5219	6.9015
512 bits	$n = 5$	0.0038	0.0008	0.0125	0.0025	0.1359	0.0272	28.5523	5.7105	0.1229	0.0246	19.8966	3.9793	0.2838	0.0568	15.2577	3.0515
	$n = 9$	0.0052	0.0007	0.0195	0.0028	0.3805	0.0544	73.5487	10.5070	0.3565	0.0509	52.0222	7.4317	0.7889	0.1127	43.0156	6.1451
	$n = 15$	0.0156	0.0010	0.0805	0.0054	3.9201	0.2613	650.2054	43.3470	3.7476	0.2498	478.2466	31.8831	7.1490	0.4766	440.0359	29.3357
1024 bits	$n = 5$	0.0040	0.0008	0.0229	0.0046	0.2284	0.0457	193.5127	38.7025	0.1876	0.0375	134.2940	26.8588	0.5368	0.1074	101.6083	20.3217
	$n = 9$	0.0082	0.0009	0.0651	0.0072	1.4559	0.1618	1015.8850	112.8761	1.3595	0.1511	725.8251	80.6472	3.1142	0.3460	625.8767	69.5419
	$n = 15$	0.0166	0.0011	0.1356	0.0090	7.6396	0.5093	4393.1295	292.8753	7.4067	0.4938	3202.2946	213.4863	14.1664	0.9444	2934.7924	195.6528

Comparing with other optimized MPC implementations A comparison with other (unverified) optimized implementations of MPC protocols, like EMP or SCALE-MAMBA, would be interesting but is outside the scope of this paper. *We stress that we do not claim to have the fastest implementation of a (proactive) secure MPC evaluator, nor is that our goal. Our goal is to demonstrate feasibility to performing computer-aided verification of such complex MPC protocols for active adversaries, and to automatically extract verified executable implementations thereof.* It would be interesting to explore how verified implementations behave in real software engineering projects, how the performance penalty induced by the verification process affects the overall performance of the system, and how much execution time developers are willing to sacrifice in order to have a more reliable system. Our performance comparison with Charm show that code obtained using our extraction approach is at least comparable with some manually implemented software, which is promising evidence that the verification overhead induced by our approach may not be as prohibitive as one may initially think. Likewise, the performance penalty of our solution is not intrinsic to the verification/extraction methodology. Some of the implementations cited above rely either on cryptographic optimizations, other (faster) protocols, faster underlying libraries (such as faster polynomial libraries) or some circuit optimization techniques. We point out that our overall verification approach can accommodate these cryptographic advances, with some implications on the security and correctness proofs. Exploring these issues is a promising avenue for future work as discussed next.

6 Related work

Previous computer-aided verification attempts paved the way for our work by demonstrating that verification of multiparty protocols is possible in EasyCrypt (even if only for smaller number of parties and/or for weaker adversaries). We not only wanted to demonstrate that verification tools like EasyCrypt have reached an interesting state of maturity, enough to formalize complex cryptographic protocols such as the ones described in this paper, but also lay the foundations that can be later used/extended to hopefully verify other cryptographic protocols in other proof environments, such as, for example, an UC proof of the BGW protocol.

The most relevant related work to ours in formal verification and high-assurance implementations of secure computation protocols is summarized in table 1. Staughton and Varia's work [68] focus on proving the adaptive, information-theoretic, honest-but-curious security of one function-specific three-party cryptographic protocol that counts the number of elements in a database in the nonprogrammable random oracle model [15]. Similarly to ours, their work also relies on EasyCrypt to check the security proofs they perform. Nevertheless, their work does not deal with generic secure computation protocols, nor with more than 3 parties.

Haagh et al. [47] verify a simple, yet very interesting and didactic, MPC protocol due to Maurer [58]. The authors followed an interesting approach to address the active security of the concrete MPC protocol: they combine non-interference techniques (which are proven to be equivalent to certain classes of cryptographic security in [5,6,47]) to prove the existence of a simulator. The non-interference approach the authors followed

is an interesting approach to reason at a higher-level, concretely in trying to understand if a simulator in fact exists for some protocol and some functionality. Additionally, they present an interesting formalization of Maurer's protocol, that can be generalized and applied to other MPC protocols. There are two main differences between our work and that of Haagh et al.: i. there is no synthesis of a high-assurance (or certified in the terminology of [4]) software to evaluate the MPC protocol; and ii. our work deals with explicit simulator definitions, thus achieving a full end-to-end EasyCrypt formalization. Finally, the verified version of Maurer's protocol in [47] is designed for general adversary structures and incurs exponential overhead (in size of shared data) in the threshold case. This limits [47] to a small number of parties.

The closest work to ours is that of Almeida et al. [4]. It develops a verified software stack for secure function evaluation by two parties in the semi-honest model. Their stack cannot handle more than two parties, nor active adversaries. The stack consists of two components: a certified compiler from C to Boolean circuits, and a high-assurance garbled circuit evaluator. The compiler proves that the output circuit is equivalent to the input C code. The Boolean circuit is then executed by a verified garbled circuit evaluator, synthesized from a formally verified EasyCrypt implementation of Yao's protocol. While this work may seem, at first sight, to have some limitations (in the number of parties and in the security model), it still represents a very interesting case of study and stepping stone in the development of verified implementations of cryptographic software.

In addition to the above, there have been impressive advances in research developing (practical) secure computation frameworks and software, either for two parties (typically based on garbled circuits) [69,57], or for multiparty using algebraic MPC approach on top of secret sharing [70,16,23], or function-specific protocols based on (fully) homomorphic encryption, or mixed versions [48,31]. Such frameworks are not directly comparable to our work because they do not claim to produce implementations of mechanically formally verified protocols nor executables thereof. We think that a very interesting future research direction is to perform computer-aided formal verification of the optimized protocols in the above frameworks and to attempt to mechanically synthesize efficient implementations thereof. We also hope that such mechanically synthesized implementations will have comparable performance to manually optimized ones.

7 Conclusions and future work

Secure and privacy-preserving protocols are an example of advanced cryptographic constructions that will probably play a crucial role in the future of our networked world and the Internet. Computer-aided verification and automated software synthesis of such complicated protocols (e.g., with proactive security guarantees against active adversaries) is achievable with EasyCrypt as our work demonstrates. As a side contribution we perform the first computer-aided verification and automated synthesis of a variant of the (fundamental) BGW secure multiparty computation (MPC) protocol for static active adversaries. We also develop a tool-chain to verifiably extract executable implementations from EasyCrypt specifications of such protocols.

Future research directions include: 1) extending our work to provide stronger security guarantees and handle more settings, e.g., adaptive active adversaries as opposed to static ones we have for BGW, dealing with dynamic groups in the standard and proactive settings, and operating over asynchronous networks; 2) performing computer-aided verification and synthesis of other (practical) MPC protocols such as SPDZ and its variations which is the basis for the efficient SCALE-MAMBA [17] MPC framework, or classic ones such as GMW [45] to deal with dishonest majorities; 3) performing full verification and synthesis of UC-secure MPC protocols and primitives to enable arbitrary compositions of them and their executables; 4) extending our protocol executables with verified libraries providing lower-level cryptographic algorithms and primitives similar to EverCrypt [18]; and 5) developing formal specification and computer-aided verifications of the underlying broadcast synchronous (or even asynchronous) communication.

[17] `https://github.com/KULeuven-COSIC/SCALE-MAMBA`
[18] `https://project-everest.github.io`

References

1. Masayuki Abe and Serge Fehr. Adaptively secure feldman vss and applications to universally-composable threshold cryptography. In Matt Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, pages 317–334, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
2. Joseph A. Akinyele, Christina Garman, Ian Miers, Matthew W. Pagano, Michael Rushanan, Matthew Green, and Aviel D. Rubin. Charm: a framework for rapidly prototyping cryptosystems. *Journal of Cryptographic Engineering*, 3(2):111–128, Jun 2013.
3. José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Verifiable side-channel security of cryptographic implementations: constant-time mee-cbc. In *23rd International Conference on Fast Software Encryption (FSE)*, pages 163–184, March 2016.
4. José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, and Vitor Pereira. A fast and verified software stack for secure function evaluation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1989–2006. ACM, 2017.
5. José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Hugo Pacheco, Vitor Pereira, and Bernardo Portela. Enforcing ideal-world leakage bounds in real-world secret sharing mpc frameworks. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 132–146. IEEE, 2018.
6. José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Hugo Pacheco, Vitor Pereira, and Bernardo Portela. Enforcing ideal-world leakage bounds in real-world secret sharing mpc frameworks. Cryptology ePrint Archive, Report 2018/404, 2018. `https://eprint.iacr.org/2018/404`.
7. Gilad Asharov and Yehuda Lindell. A full proof of the bgw protocol for perfectly-secure multiparty computation. Cryptology ePrint Archive, Report 2011/136, 2011. `https://eprint.iacr.org/2011/136`.
8. Michael Backes, Aniket Kate, and Arpita Patra. Computational verifiable secret sharing revisited. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, pages 590–609, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
9. Michael Backes, Matteo Maffei, and Esfandiar Mohammadi. Computationally sound abstraction and verification of secure multi-party computations. In *FSTTCS*, 2010.
10. Cécile Baritel-Ruet, François Dupressoir, Pierre-Alain Fouque, and Benjamin Grégoire. Formal security proof of cmac and its variants. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 91–104. IEEE, 2018.
11. Joshua Baron, Karim Eldefrawy, Joshua Lampkins, and Rafail Ostrovsky. How to withstand mobile virus attacks, revisited. In *PODC*, pages 293–302. ACM, 2014.
12. Joshua Baron, Karim Eldefrawy, Joshua Lampkins, and Rafail Ostrovsky. Communication-optimal proactive secret sharing for dynamic groups. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *Applied Cryptography and Network Security*, pages 23–41, Cham, 2015. Springer International Publishing.
13. Joshua Baron, Karim Eldefrawy, Joshua Lampkins, and Rafail Ostrovsky. Communication-optimal proactive secret sharing for dynamic groups. In *ACNS*, volume 9092 of *LNCS*, pages 23–41. Springer, 2015.
14. Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella-Béguelin. Probabilistic relational verification for cryptographic implementations. In *POPL*, 2014. To appear.
15. Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, pages 62–73, New York, NY, USA, 1993. ACM.
16. Assaf Ben-David, Noam Nisan, and Benny Pinkas. Fairplaymp: a system for secure multi-party computation. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 257–266. ACM, 2008.
17. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the 20th Annual Symposium on Theory of Computing*, pages 1–10. ACM, 1988.
18. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10. ACM, 1988.
19. Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jianyang Pan, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Béguelin, and Jean Karim Zinzindohoué. Implementing and proving the tls 1.3 record layer. Cryptology ePrint Archive, Report 2016/1178, 2016. `http://eprint.iacr.org/2016/1178`.
20. Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *IEEE S&P*, 2013.

21. G. R. Blakley. Safeguarding cryptographic keys. *Proc. of AFIPS National Computer Conference*, 48:313–317, 1979.
22. Dan Bogdanov, Peeter Laud, Sven Laur, and Pille Pullonen. From input private to universally composable secure multi-party computation primitives. In *Proceedings of the 27th Computer Security Foundations Symposium*, pages 184–198. IEEE, 2014.
23. Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.
24. Sally Browning and Philip Weaver. *Designing Tunable, Verifiable Cryptographic Hardware Using Cryptol*, pages 89–143. 01 2010.
25. Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *ACM Conference on Computer and Communications Security*, pages 88–97, 2002.
26. Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 88–97, New York, NY, USA, 2002. ACM.
27. B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 383–395, Oct 1985.
28. Véronique Cortier, Constantin Catalin Dragan, François Dupressoir, and Bogdan Warinschi. Machine-checked proofs for electronic voting: privacy and verifiability for belenios. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 298–312. IEEE, 2018.
29. Morten Dahl and Ivan Damgård. Universally composable symbolic analysis for two-party protocols based on homomorphic encryption. In *EUROCRYPT*, 2014.
30. Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. Confidential benchmarking based on multiparty computation. In *Proceedings of the 20th International Conference on Financial Cryptography and Data Security*, pages 169–187. Springer, 2016.
31. Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.
32. Shlomi Dolev, Karim Eldefrawy, Joshua Lampkins, Rafail Ostrovsky, and Moti Yung. Proactive secret sharing with a dishonest majority. In Vassilis Zikas and Roberto De Prisco, editors, *Security and Cryptography for Networks*, pages 529–548, Cham, 2016. Springer International Publishing.
33. Shlomi Dolev, Karim Eldefrawy, Joshua Lampkins, Rafail Ostrovsky, and Moti Yung. Proactive secret sharing with a dishonest majority. In *SCN*, volume 9841 of *LNCS*, pages 529–548. Springer, 2016.
34. Yael Ejgenberg, Moriya Farbstein, Meital Levy, and Yehuda Lindell. Scapi: The secure computation application programming interface. Cryptology ePrint Archive, Report 2012/629, 2012. `https://eprint.iacr.org/2012/629`.
35. Karim Eldefrawy, Rafail Ostrovsky, Sunoo Park, and Moti Yung. Proactive secure multiparty computation with a dishonest majority. In *Proceedings of the Eleventh Conference on Security and Cryptography for Networks*, pages 200–215, 01 2018.
36. Karim Eldefrawy and Vitor Pereira. A high-assurance evaluator for machine-checked secure multiparty computation. Cryptology ePrint Archive, Report 2019/922, 2019. `https://eprint.iacr.org/2019/922`.
37. P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 427–438, Oct 1987.
38. Jean-Christophe Filliâtre. One logic to use them all. In *24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Artificial Intelligence*, pages 1–20, Lake Placid, USA, June 2013. Springer.
39. Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.
40. Matthias Fitzi, Juan Garay, Shyamnath Gollakota, C. Pandu Rangan, and Kannan Srinathan. Round-optimal and efficient verifiable secret sharing. In Shai Halevi and Tal Rabin, editors, *Theory of Cryptography*, pages 329–342, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
41. Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In *ACM CCS*, 2011.
42. Matthew Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 699–710, New York, NY, USA, 1992. ACM.

43. Rosario Gennaro, Yuval Ishai, Eyal Kushilevitz, and Tal Rabin. The round complexity of verifiable secret sharing and secure multicast. In *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing*, STOC '01, pages 580–589, New York, NY, USA, 2001. ACM.
44. Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified vss and fast-track multiparty computations with applications to threshold cryptography. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '98, pages 101–111, New York, NY, USA, 1998. ACM.
45. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229, 1987.
46. Helene Haagh, Aleksandr Karbyshev, Sabine Oechsner, Bas Spitters, and Pierre-Yves Strub. Computer-aided proofs for multiparty computation with active security. pages 119–131, 07 2018.
47. Helene Haagh, Aleksandr Karbyshev, Sabine Oechsner, Bas Spitters, and Pierre-Yves Strub. Computer-aided proofs for multiparty computation with active security. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 119–131. IEEE Computer Society, 2018.
48. Wilko Henecka, Stefan K ögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. Tasty: Tool for automating secure two-party computations. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 451–462, New York, NY, USA, 2010. ACM.
49. Wilko Henecka, Stefan KÃ¶gl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. Tasty: Tool for automating secure two-party computations. volume 2010, pages 451–462, 01 2010.
50. Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *CRYPTO*, pages 339–352, 1995.
51. Amir Herzberg, Stanisław Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In Don Coppersmith, editor, *Advances in Cryptology — CRYPT0' 95*, pages 339–352, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.
52. Martin Hirt, Christoph Lucas, and Ueli Maurer. A dynamic tradeoff between active and passive corruptions in secure multi-party computation. In *CRYPTO (2)*, volume 8043 of *LNCS*, pages 203–219. Springer, 2013.
53. Jonathan Katz, Chiu-Yuen Koo, and Ranjit Kumaresan. Improving the round complexity of vss in point-to-point networks. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming*, pages 499–510, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
54. Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is ssl?). In Joe Kilian, editor, *Advances in Cryptology — CRYPTO 2001*, pages 310–331, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
55. Ranjit Kumaresan, Arpita Patra, and C. Pandu Rangan. The round complexity of verifiable secret sharing: The statistical case. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, pages 431–447, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
56. Baiyu Li and Daniele Micciancio. Symbolic security of garbled circuits. pages 147–161, 07 2018.
57. Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In Matt Blaze, editor, *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 287–302. USENIX, 2004.
58. Ueli M. Maurer. Secure multi-party computation made simple. *Discrete Applied Mathematics*, 154(2):370–381, 2006.
59. Rafail Ostrovsky and Moti Yung. How to withstand mobile virus attacks (extended abstract). In *PODC*, pages 51–59. ACM, 1991.
60. Mário José Parreira Pereira. *Tools and Techniques for the Verification of Modular Stateful Code*. PhD thesis, Paris Saclay, 2018.
61. Arpita Patra, Ashish Choudhary, Tal Rabin, and C. Pandu Rangan. The round complexity of verifiable secret sharing revisited. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009*, pages 487–504, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
62. Torben Pryds Pedersen. A threshold cryptosystem without a trusted party. In *Proceedings of the 10th Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'91, pages 522–526, Berlin, Heidelberg, 1991. Springer-Verlag.
63. Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 129–140, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.
64. Aseem Rastogi, Nikhil Swamy, and Michael Hicks. Wys*: A verified language extension for secure multi-party computations. *CoRR*, abs/1711.06467, 2017.
65. David Schultz. *Mobile Proactive Secret Sharing*. PhD thesis, Massachusetts Institute of Technology, 2007.

66. Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.
67. Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.
68. Alley Stoughton and Mayank Varia. Mechanizing the proof of adaptive, information-theoretic security of cryptographic protocols in the random oracle model. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pages 83–99. IEEE Computer Society, 2017.
69. Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. Faster secure two-party computation in the single-execution setting. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part III*, volume 10212 of *Lecture Notes in Computer Science*, pages 399–424, 2017.
70. Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 39–56. ACM, 2017.
71. Theodore M. Wong, Chenxi Wang, and Jeannette M. Wing. Verifiable secret redistribution for archive system. In *IEEE Security in Storage Workshop*, pages 94–106. IEEE Computer Society, 2002.
72. Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE Computer Society, 1982.
73. Lidong Zhou, Fred B. Schneider, and Robbert van Renesse. Apss: proactive secret sharing in asynchronous systems. *ACM Trans. Inf. Syst. Secur.*, 8(3):259–286, 2005.

Appendix A Reasoning About Polynomials

We had to formalize a verified library to reason about polynomials because several protocols rely heavily on polynomials and operations over them. This was a critical step in our proofs, otherwise we would have needed to use a non-verified polynomial library to perform the desired operations, which would increase our trusted code base.

We are interested in polynomials over finite fields in this work. However, we developed a general polynomials library, that could be reused to define multiple instances of polynomials. We started the development of our polynomials library by first defining an abstract definition for it. This abstract definition provides an interface for concrete realizations of polynomials and defines the coefficient type, polynomial evaluation, the *zero* and *one* polynomials, polynomial degree, addition, multiplication, unary minus and interpolation. We fix the type of the polynomials to be a list of monomials, which induces a possible equality class to be the equality between lists. Such equality class is too strong and would force the definition of complicated operators to perform addition and multiplication, thus inducing a significant performance penalty. We fix the equality class to be the equality of the evaluation of two polynomials in the same points, and use this equality class to define the subsequent axioms and lemmas around the polynomial operations mentioned above. This abstract polynomial interface is reusable and can be applied to other domains.

We then instantiate the type of the coefficients to be the same type of elements in a finite field and define all polynomial operations in the expected, classical way. Properties such as commutativity and associativity of polynomial addition and multiplication were easily proven by relying on the same properties verified in finite field operations. We also provide a formalization for polynomial interpolation based on the Lagrange interpolation, which makes use of a linear combination of Lagrange basis polynomials. Note that polynomial interpolation is important in our work, since it allows secret reconstruction inside the reconstruct protocol and also recovering parties to successfully recover shares.

In order to define polynomials, we make use of EasyCrypt's record system as shown in Figure 25. We see polynomials as lists of monomials, which are a record with two fields: a *coefficient* (an element of a finite field) and an *exponent* (an integer).

```
type coefficient = t.
type exponent = int.
type monomial = {
  coef : coefficient;
  expo : exponent
}.
type polynomial = monomial list.
```

Fig. 25: Polynomial type definition

Next, our polynomial library defines evaluation functions, one for monomial evaluation and other for polynomials that is basically multiple applications of the first one. Polynomial evaluation defines our equality class. All this definitions can be found in Figure 26. We also include in this Figure a polynomial *membership* test: a point is the polynomial if the evaluation of the polynomial at the abscissa is equal to the value of the ordinate.

We are also interested in having a *zero* (i.e. a polynomial that always evaluate to zero) and a *one* polynomial (i.e. a polynomial that always evaluate to one) so that we are able to define algebraic properties around polynomial operations. Those polynomials are depicted in Figure 27. Note that `zero` could also be defined based on `mzero` but defining it as an empty list simplifies proofs because it allows one to prove polynomial properties based on list induction.

Figure 28 sketches the EasyCrypt definition of polynomial arithmetic. Every arithmetic operation is defined in the same, mechanical way: 1) we start by defining that operation in terms of monomials (`madd`, `mmul`

```
op meval (x:coefficient) (m : monomial) = m.`coef * (x ^ m.`expo).
op eval (x:coefficient) p =
  with p = [] => F.zero
  with p = m :: p' => meval x m + (eval x p').
op (==) p1 p2 = forall x, eval x p1 = eval x p2.
op mem (pt : (coeficient * coeficient)) p = eval (fst pt) p = (snd pt).
```

Fig. 26: Polynomial evaluation and equality class

```
op mzero = {| coef = F.zero; expo = 1 |}.
op zero : polynomial = [].
op mone = {| coef = F.one; expo = 0 |}.
op one = [mone].
```

Fig. 27: *Zero* and *one* polynomials

and `mumin`); 2) then we define it in terms of a monomial and a polynomial (`mpadd` and `mpmul`); and finally 3) we define the polynomial operation based on the previous two (`add`, `mul` and `umin`) The reason why we implement polynomial operations as such lies with proof simplification. Proving properties related to monomial operations is much more easy than proving them applied to polynomials. Yet, since polynomial operations are defined based on the monomials' ones, it is easy to propagate results obtained at monomial level to polynomials. Additionally, interesting arithmetic properties (such as commutativity or associativity) of polynomial operations are easily proven by relying on the same properties of their underlying coefficients.

The last operations delivered by our polynomial interface is Lagrange interpolation. Lagrange interpolation allows to reconstruct a $d-1$ degree polynomial based on d points. Briefly, it works by computing *bases* based on the abscissa values, which are then multiplied by the ordinate values. As part of this project, we provide two different interpolation functions:

- `interpolate` - taking as input a set of points and a some x value, returns the evaluation of the interpolated polynomial on x
- `interpolate_poly` - taking as input a set of points, returns the interpolated polynomial.

EasyCrypt polynomial interpolation is presented in Figure 29.

Appendix B Secret sharing security definitions

Figure 30 defines passive security for a secret sharing scheme. Theory HBCSecretSharingSchemeSecurity is parameterized by a secret sharing scheme, making it modular enough to be reused when representing security for multiple secret sharing schemes. It starts by first defining an abstract random generator type that will be used to feed the explicit randomness needed to execute probabilistic algorithms such as share. This random generator is abstract because the secret sharing scheme is also abstract and thus there is no information about the type of randomness involved. Notwithstanding, the instantiation step will make it concrete. Next, there is the definition of oracles. For the particular case of semi-honest security, we are only interested in providing the adversary with an oracle to corrupt parties. An adversary attacking the scheme (parameterized by the oracles) should, therefore, have two procedures:

- choose - that creates a query of two secrets
- guess - that tries to guess which secret was the origin of the received shares

The security experience follows naturally. The adversary chooses two secrets and tries to distinguish between the set of shares he receives. He wins the game if he is able to do some with probability 1.

```
op madd m1 m2 = {| coef = m1.‘coef + m2.‘coef; expo = m1.‘expo |}.
op mpadd (m : monomial) p =
  with p = [] => [m]
  with p = m’ :: p’ =>
    if m.‘expo = m’.‘expo then madd m m’ :: p’
    else
      if m’.‘expo < m.‘expo then m :: p
      else m’ :: mpadd m p’.
op add (p1 p2 : polynomial) =
  with p1 = [], p2 = [] => []
  with p1 = m1 :: p1’, p2 = [] => p1
  with p1 = [], p2 = m2 :: p2’ => p2
  with p1 = m1 :: p1’, p2 = m2 :: p2’ =>
    if m1.‘expo = m2.‘expo then madd m1 m2 :: add p1’ p2’
    else
      if m1.‘expo < m2.‘expo then m2 :: add p1 p2’
      else m1 :: add p1’ p2.
op mmul m1 m2 =
  if m1 = mzero \/ m2 = mzero then mzero
  else {| coef = m1.‘coef * m2.‘coef; expo = m1.‘expo + m2.‘expo |}.
op mpmul m p =
  with p = [] => []
  with p = m’ :: p’ => mpadd (mmul m m’) (mpmul m p’).
op mul p1 p2 =
  with p1 = [] => []
  with p1 = m :: p1’ => add (mpmul m p2) (mul p1’ p2).
op mumin m = {| coef = - m.‘coef; expo = m.‘expo |}.
op umin p =
   with p = [] => []
   with p = m :: p’ => mumin m :: umin p’.
```

Fig. 28: Polynomial arithmetic

```
op basis_loop (x : t) (xmx : t) (xm : t list) : t list =
  with xm = [] => []
  with xm = y :: ys =>
    if y <> xmx then
      ((x - y) / (xmx - y)) :: basis_loop x xmx ys
    else basis_loop x xmx ys.
op basis (x xmx : t) (xm : t list) =
  foldr (fun (x y : t) => x * y) F.one (basis_loop x xmx xm).
op interpolate_loop (x : t) (xm : t list) (pm : (t * t) list) =
  with pm = [] => []
  with pm = y :: ys =>
    ((basis x (fst y) xm) * (snd y)) :: interpolate_loop x xm ys.
op interpolate (x : t) (pm : (t * t) list) =
  let xm = map fst pm in
  let bs = interpolate_loop x xm pm in
  foldr (fun (x y : t) => x + y) F.zero bs.
```

Fig. 29: Polynomial Lagrange interpolation

```
theory HBCSecretSharingSchemeSecurity.
  clone import SecretSharingScheme.
  module type Rand_t = {
    proc gen() : rand_t
  }.
  module type Oracles_t = {
    proc corrupt(pid : p_id_t) : unit
  }.
  ...
  module type Adv_t (O : Oracles_t) = {
    proc choose() : secret_t * secret_t
    proc guess(ss : shares_t) : bool
  }.
  module Game (R : Rand_t) (A : Adv_t) = {
    module O = Oracles
    module A = A(O)
    proc main() : bool = {
      ...
      b <\$ {0,1}; r <@ R.gen(); O.init();
      (s0,s1) <@ A.choose();
      ss <- share r (b ? s1 : s0);
      ss <- get_corrupted O.corrupted ss;
      b' <@ A.guess(ss);
      return (b = b');
    }
  }.
end HBCSecretSharingSchemeSecurity.
```

Fig. 30: Honest-but-curious (HBC) security for secret sharing

In order to define integrity of shares, we used the security notion depicted in Figure 31. To win the security game, an adversary needs to provide some forgery of shares that were not obtained via an honest execution of the secret sharing scheme. We model the entire behaviour of the security experience in the adversary oracles by giving him access to a *share* oracle - that provides the adversary honestly generated shares -, and with a *forge* oracle - which the adversary can use in order to test if some shares are a valid forgery.

The reason why we are modeling the security goal inside an adversary oracle instead of the expected way (the adversary only have access to a *share* oracle and it outputs shares that are going to be tested for forge validity) is because it greatly simplifies the composition proof with an honest-but-curious secret sharing scheme.

The security of a commitment scheme is presented in Figure 32. It is a very similar security definition for other integrity or authenticity schemes such as MAC schemes. An adversary can require commits and also can check if some commitment is valid for some message. It will then try to forge a message/commitment pair that verifies but that was not obtained via an honest execution of the commitment scheme.

Finally, we define malicious security for secret sharing schemes as shown in Figure 33. In this security experience, the adversary will try to break either the integrity of shares or the indistinguishability of them. With that purpose, it is given access to a *reconstruct* oracle that provide it with secrets reconstructed using the desired set of shares. A malicious (verifiable) secret sharing scheme can then be obtained by composing an honest-but-curious secret sharing scheme with an unforgeable commitment scheme.

```
theory INTSecretSharingSchemeSecurity.
  clone import SecretSharingScheme.
  module type Rand_t = {
    proc gen() : rand_t
  }.
  module type Oracles_t = {
    proc share(s : secret_t) : shares_t option
    proc forge(ss : shares_t) : unit
  }.
  module type Adv_t(O : Oracles_t) = {
    proc main() : unit
  }.
  ...
  module Game (R : Rand_t) (A : Adv_t) = {
    module O = Oracles
    module A = A(O)
    proc main() : bool = {
      ...
      r <@ R.gen(); O.init(r);
      A.main();
      return (O.forgery);
    }
  }.
end INTSecretSharingSchemeSecurity.
```

Fig. 31: Integrity security for secret sharing

Appendix C MPC security definitions

The *private* (active) security definition for MPC protocols can be found in Figure 34. The security experience works by phases:

- The initial phase (*initial* in Figure 34) is first initialized with the protocol inputs and randomness, as well as with an empty corrupted set.
- Before the execution of each phase, the adversary has the ability to either change the input with which a party is going to execute the stage of the protocol or simply abandon it.
- The result of the phase execution are then used as input to the next phase of the protocol.

In the ideal scenario, the experience will run a simulator using some auxiliary input (which we depict as *leak_t*) and the input shares of corrupted parties. This simulator will need to be able to produce random coins and communication traces that have the same probability distribution as the ones outputed by the real protocol execution. For this particular case of *private* security, we want the simulator to also be able to simuate the output share of corrupted parties.

Random security definition in Figure 35 is defined in a very similar way to *private* security. In fact, the only differences are the protocol input type - each party will only have a share as input - and the type of the simulator - which will now need to produce random coins and conversation traces based on auxiliary input, corrupted input shares and corrupted output shares.

Similarly to functionalities, *proactive* security can be seen as a special case of a re-randomization protocol like refresh with the caveat that it undermines previous shares of recovered parties. At a high level, this means that, after the execution of the protocol, the recovering party will have a *good* share even if it started with a corrupt one. The definition of *proactive* security can be found in Figure 36.

```
theory CommitmentShemeSecurity.
  clone import CommitmentScheme.
  ...
  module type Rand_t = {
    proc gen() : rand_t
  }.
  module type Oracles_t = {
    proc mac(m : msg_t) : commit_t
    proc verify(m : msg_t, t : commit_t) : bool
    proc forge(m : msg_t, t : commit_t) : unit
  }.
  module type Adv_t(O : Oracles_t) = {
    proc main() : unit
  }.
  ...
  module Game(R : Rand_t, A : Adv_t) = {
    module O = Oracles
    module A = A(O)
    proc main(): bool = {
      ...
      r <@ R.gen(); O.init(r);
      A.main();
      return (O.forgery);
    }
  }.
end CommitmentShemeSecurity.
```

Fig. 32: Unforgeability for commitment schemes

Appendix D MPC concrete protocols

This Appendix will focus on the presentation of several EasyCrypt implementations of MPC protocols, both in the passive and malicious setting. Due to space constrains, we will focus on the functional behaviour of protocols.

An honest-but-curious version of the addition protocol can be found in Figure 37. It is a very simple protocol, since it is composed only by local operations. The only difference between the passive version and the malicious one (Figure 38) is the computation of new homomorphic commitments to the new shares.

Figures 39 and 40 represent the passive and active versions of the refresh protocol, respectively. The two main differences between the two rely on the use of homomorphic commitments to ensure share integrity: first, parties will produce sharings of zero using a verifiable secret sharing scheme and then, shares will be checked for consistency during the addition step. Note that, besides being a simple modification, it induces a significant performace penalty as it requires the computation of several field and cyclic group exponentiations.

```
theory MALSecretSharingSchemeSecurity.
  clone import SecretSharingScheme.
  ...
  module type Rand_t = {
    proc gen() : rand_t
  }.
  module type Oracles_t = {
    proc corrupt(pid : p_id_t) : unit
    proc reconstruct(ss : shares_t) : secret_t option
  }.
  module type Adv_t (O : Oracles_t) = {
    proc choose() : secret_t * secret_t
    proc guess(ss : shares_t) : bool
  }.
  module Game (R : Rand_t) (A : Adv_t) = {
    module O = Oracles
    module A = A(O)
    proc main() : bool = {
      ...
      b <\$ {0,1}; O.init(); r <@ R.gen();
      (s0, s1) <@ A.choose();
      ss <- share r (b ? s1 : s0);
      ss <- get_corrupted O.corrupted ss;
      b' <@ A.guess(ss);
      return (b = b');
    }
  }.
end MALSecretSharingSchemeSecurity.
```

Fig. 33: Malicious security for secret sharing

An honest-but-curious version of the recover protocol can be obtained via Figure 41. This version is simpler than the malicious one presented in Section 3.3 since it does not deal with share integrity verification. In fact, the only difficulty in implementing this protocol lies on the polynomial interpolation, since every other operation is a combination of the share and addition protocols.

We end with the presentation of protocols to perform multiplication, both passive - Figure 42 -, and malicious - Figure 43. Multiplication is done following directions pointed by Asharov and Lindell in [7]. Briefly, multiplication works by parties first multiplying their shares and then subsharing them in order to perform polynomial degree reduction. For the malicious protocol, parties need to cope with possible modification of subshares and thus there is the need to use an error correction algorithm upon receiving all sub-shares.

```
theory ProtocolPrivateSecurity.
  ...
  module Game (R : Rand_t, Z : Environment_t, A : Adversary_t, S : Simulator_t) = {
    module O = Oracles
    module A = A(O)

    b <\${0,1};
    inps <@ Z.choose();
    r <@ R.gen(inps);
    C.init(inps);
    if (valid_inputs inps /\ valid_rands r inps) {
      if (b) {
        A.run();
        (cc,yy) <- P.prot r inps;
        cc <- filter_corrupt_convs cc C.corrupt;
        b' <@ Z.guess(poutput2foutput yy, filter_corrupt_outputs yy C.corrupt, cc);
      }
      else {
        A.run();
        finp <- pinput2finput inps;
        y <- F.f finp;
        l <- F.ϕ finp;
        (yy, cc) <@ S.simm(l, filter_corrupt_inputs inps C.corrupt, C.corrupt);
        b' <@ Z.guess(y, yy, cc);
      }
    }
    else {
      b <\$ {0,1}
    }

    return b';
  }.
end ProtocolPrivateSecurity.
```

Fig. 34: *Private* security

```
theory ProtocolRandomSecurity.
  ...
  module Game (R : Rand_t, Z : Environment_t, A : Adversary_t, S : Simulator_t) = {
    module O = Oracles
    module A = A(O)

    b <\${0,1};
    inps <@ Z.choose();
    r <@ R.gen(inps);
    C.init(inps);
    if (valid_inputs inps /\ valid_rands r inps) {
      if (b) {
        A.run();
        (cc,yy) <- P.prot r inps;
        b' <@ Z.guess(filter_corrupt_outputs yy C.corrupt, filter_corrupt_convs cc C.corrupt);
      }
      else {
        finp <- pinput2finput inps;
        y <- F.f r finp;
        l <- F.ϕ r finp;
        cc <@ S.simm(l, filter_corrupt_inputs inps C.corrupt, filter_corrupt_foutput y C.corrupt, C.
            corrupt);
        b' <@ Z.guess(filter_corrupt_outputs (foutput2poutput y) C.corrupt, cc);
      }
    }
    else {
      b <\$ {0,1}
    }

    return b';
  }.
end ProtocolRandomSecurity.
```

Fig. 35: *Random* security

```
theory ProtocolProactiveSecurity.
  ...
  module Game (R : Rand_t, Z : Environment_t, A : Adversary_t, S : Simulator_t) = {
    module O = Oracles
    module A = A(O)

    b <\${0,1};
    inps <@ Z.choose();
    r <@ R.gen(inps);
    C.init(inps);
    if (valid_inputs inps /\ valid_rands r inps) {
      if (b) {
        A.run();
        (cc,yy) <- P.prot r inps;
        b' <@ Z.guess(filter_corrupt_outputs yy C.corrupt, filter_corrupt_convs cc C.corrupt);
      }
      else {
        finp <- pinput2finput inps;
        y <- F.f r finp;
        l <- F.ϕ r finp;
        cc <@ S.simm(l, filter_corrupt_inputs inps C.corrupt, filter_corrupt_foutput y C.corrupt, C.
            corrupt);
        b' <@ Z.guess(filter_corrupt_outputs (foutput2poutput y) C.corrupt, cc);
      }
    }
    else {
      b <\$ {0,1}
    }

    return b';
  }.
end ProtocolProactiveSecurity.
```

Fig. 36: *Proactive* security

```
theory HBCAdditionProtocol.
  ...
  type p_id_t = p_id_t.
  type input_t = share_t * share_t.
  type inputs_t = (p_id_t * input_t) list.
  type output_t = share_t.
  type outputs_t = (p_id_t * output_t) list.
  type rand_t = unit.
  type rands_t = (p_id_t * rand_t) list.
  type conv_t = unit.
  type convs_t = (p_id_t * conv_t) list.
  op pexec (pi : input_t) : output_t =
    let (xi,yi) = pi in
    (xi + yi).
  op prot (rs : rands_t) (pis : inputs_t) : (convs_t * outputs_t) option =
    let os = map (fun pi => (fst pi, pexec (snd pi))) pis in
    let cs = map (fun pi => (fst pi, ())) pis in
    Some (cs, os).
  clone import Protocol as HBCAddition with
    ...
end HBCAdditionProtocol.
```

Fig. 37: Honest-but-curious addition protocol

```
theory MALAdditionProtocol.
  ...
  type p_id_t = p_id_t.
  type input_t = share_t * share_t.
  type inputs_t = (p_id_t * input_t) list.
  type output_t = share_t.
  type outputs_t = (p_id_t * output_t) list.
  type rand_t = unit.
  type rands_t = (p_id_t * rand_t) list.
  type conv_t = unit.
  type convs_t = (p_id_t * conv_t) list.
  op update_commits (c c' : commit_t) : commit_t =
    (c.`1 + c'.`1, map2 (fun x y => (fst x, CyclicGroup.( * ) (snd x) (snd y))) c.`2 c'.`2).
  op pexec (pi : input_t) : output_t =
    let (xi,yi) = pi in
    (xi.`1 + yi.`1, update_commits xi.`2 yi.`2).
  op prot (rs : rands_t) (pis : inputs_t) : (convs_t * outputs_t) option =
    let os = map (fun pi => (fst pi, pexec (snd pi))) pis in
    let cs = map (fun pi => (fst pi, ())) pis in
    Some (cs, os).
  clone import Protocol as MalAddition with
    ...
end MALAdditionProtocol.
```

Fig. 38: Malicious addition protocol

```
theory HBCRefreshProtocol.
  ...
  type p_id_t = p_id_t.
  op p_id_set = ShamirSS.p_id_set.
  type input_t = share_t.
  type inputs_t = (p_id_t * input_t) list.
  type output_t = share_t.
  type outputs_t = (p_id_t * output_t) list.
  type rand_t = ShamirSS.rand_t.
  type rands_t = (p_id_t * rand_t) list.
  type conv_t = share_t list.
  type convs_t = (p_id_t * conv_t) list.
  type pstate1_t = ShamirSS.shares_t.
  op pstage1 (r : rand_t) : pstate1_t =
    ShamirSS.share r F.zero.
  op pstage2 (i : input_t) (ss : share_t list) : output_t =
    foldr (fun (x : share_t) (acc : share_t) => AdditionProtocol.pexec (x, acc)) i ss.
  op prot (rs : rands_t) (iss : inputs_t) : (convs_t * outputs_t) option =
    let pst1 = map (fun pid => (pid, pstage1 (oget (assoc rs pid)))) p_id_set in
    let cs = map (fun pid => (pid, map (fun ss => snd ss) (oget (assoc pst1 pid)))) p_id_set in
    let sss = map (fun pid => (pid, map (fun idss => oget (assoc (snd idss) pid)) pst1)) p_id_set
        in
    let os = map (fun pid => (pid, pstage2 (oget (assoc iss pid)) (oget (assoc sss pid)))) p_id_set
         in
    let cs = map (fun pid => (pid, oget (assoc cs pid) ++ (oget (assoc sss pid)))) p_id_set in
    Some (cs,os).
  clone import Protocol as HBCRefresh with
    ...
end HBCRefreshProtocol.
```

Fig. 39: Honest-but-curious refresh protocol

```
theory MALRefreshProtocol.
  ...
  type p_id_t = p_id_t.
  op p_id_set = p_id_set.
  type input_t = share_t.
  type inputs_t = (p_id_t * input_t) list.
  type output_t = share_t.
  type outputs_t = (p_id_t * output_t) list.
  type rand_t = rand_t.
  type rands_t = (p_id_t * rand_t) list.
  type conv_t = share_t list.
  type convs_t = (p_id_t * conv_t) list.
  type pstate1_t = shares_t.
  op pstage1 (r : rand_t) : pstate1_t =
    share r F.zero.
  op pstage2 (pid : p_id_t) (i : input_t) (ss : share_t list) : output_t =
    foldr (fun (x : share_t) (acc : share_t) =>
           if verify (pid, fst x) (snd x) then
             AdditionProtocol.pexec (x,acc)
           else acc) i ss.
  op prot (rs : rands_t) (iss : inputs_t) : (convs_t * outputs_t) option =
    let pst1 = map (fun pid => (pid, pstage1 (oget (assoc rs pid)))) p_id_set in
    let cs = map (fun pid => (pid, map (fun ss => snd ss) (oget (assoc pst1 pid)))) p_id_set in
    let sss = map (fun pid => (pid, map (fun idss => oget (assoc (snd idss) pid)) pst1)) p_id_set
        in
    let os = map (fun pid => (pid, pstage2 pid (oget (assoc iss pid)) (oget (assoc sss pid))))
        p_id_set in
    let cs = map (fun pid => (pid, oget (assoc cs pid) ++ (oget (assoc sss pid)))) p_id_set in
    Some (cs,os).
  clone import Protocol as MalRefresh with
    ...
end MALRefreshProtocol.
```

Fig. 40: Malicious refresh

```
theory HBCRecoverProtocol.
  ...
  type p_id_t = p_id_t.
  op p_id_set = ShamirSS.p_id_set.
  type input_t = p_id_t * share_t.
  type inputs_t = (p_id_t * input_t) list.
  type output_t = share_t.
  type outputs_t = (p_id_t * output_t) list.
  type rand_t = ShamirSS.rand_t.
  type rands_t = (p_id_t * rand_t) list.
  type conv_t = share_t list.
  type convs_t = (p_id_t * conv_t) list.
  type pstate1_t = ShamirSS.shares_t.
  op pstage1 (r : rand_t) : pstate1_t =
    ShamirSS.share r F.zero.
  op pstage2 (i : share_t) (ss : share_t list) : output_t =
    foldr (fun (x : share_t) (acc : share_t) => AdditionProtocol.pexec (x, acc)) i ss.
  op rparty (pid : p_id_t) (ss : (p_id_t * share_t) list) : output_t =
    interpolate pid ss.
  op prot (rs : rands_t) (iss : inputs_t) : (convs_t * outputs_t) option =
    let rp = fst (snd (head witness iss)) in
    let nrp = rem rp p_id_set in
    let iss = map (fun pidi => let (pid,i) = pidi in (pid, snd i)) iss in
    let pst1 = map (fun pid => (pid, pstage1 (oget (assoc rs pid)))) nrp in
    let cs = map (fun pid => (pid, map (fun ss => snd ss) (oget (assoc pst1 pid)))) nrp in
    let sss = map (fun pid => (pid, map (fun idss => oget (assoc (snd idss) pid)) pst1)) nrp in
    let os = map (fun pid => (pid, pstage2 (oget (assoc iss pid)) (oget (assoc sss pid)))) nrp in
    let cs = map (fun pid => (pid, oget (assoc cs pid) ++ (oget (assoc sss pid)))) nrp in
    let orp = rparty rp os in
    let crp = map (fun x => snd x) os in
    Some ((rp, crp) :: cs, (rp, orp) :: os).
  clone import Protocol as HBCRecover with
    ...
end HBCRecoverProtocol.
```

Fig. 41: Honest-but-curious recover

```
theory HBCMultiplicationProtocol.
  type p_id_t = p_id_t.
  type p_id_set = p_id_set.
  type input_t = share_t * share_t.
  type inputs_t = (p_id_t * input_t) list.
  type output_t = share_t.
  type outputs_t = (p_id_t * output_t) list.
  type rand_t = rand_t.
  type rands_t = (p_id_t * rand_t) list.
  type conv_t = share_t list.
  type convs_t = (p_id_t * conv_t) list.
  type bdcst_1 = shares_t.
  type state1 = share_t.
  op pstage1 (r : rand_t) (i : input_t) : conv_t * state1 * bdcst_1 =
    let mi = i.`1 * i.`2 in
    let ss = ShamirSS.share r mi in
    (unzip2 ss, mi, ss).
  op pstage2 (pid : p_id_t) (c : conv_t) (st : state1) (ss : share_t list) : conv_t * output_t =
    let sum = summation (map (fun sh, basis pid sh ss) ss) in
    (c++ss,sum).
  op prot (rs : rands_t) (iss : inputs_t) : (convs_t * outputs_t) option =
    let pid_set = unzip1 iss in
    let stage1 = map (fun pid => let p1 = pstage1 (oget (assoc rs pid)) (oget (assoc iss pid)) in
        ((pid, p1.`1),(pid, p1.`2), p1.`3)) pid_set in
    let cs = unzip13 stage1 in let sts = unzip23 stage1 in let bs = unzip33 stage1 in
    let stage2 = map (fun pid => let p2 = pstage2 pid (oget (assoc cs pid)) (oget (assoc sts pid))
        (get_all_assoc pid bs) in ((pid, p2.`1),(pid, p2.`2))) pid_set in
    Some (unzip1 stage2, unzip2 stage2).
  clone import Protocol as HBCMultiplication with
    ...
end HBCMultiplicationProtocol.
```

Fig. 42: Honest-but-curious mul protocol

```
theory MALMultiplicationProtocol.
  type p_id_t = p_id_t.
  type p_id_set = p_id_set.
  type input_t = (share_t * share_t) * t list.
  type inputs_t = (p_id_t * input_t) list.
  type output_t = share_t.
  type outputs_t = (p_id_t * output_t) list.
  type rand_t = rand_t.
  type rands_t = (p_id_t * rand_t) list.
  type conv_t = share_t list.
  type convs_t = (p_id_t * conv_t) list.
  op pstage1 (i : input_t) : t = (fst (fst (fst i))) * (fst (snd (fst i))).
  type pstate2 = shares_t
  op pstage2 (pid : p_id_t) (r : rand_t) (i : t) : pstate2 = share r i.
  op pstage3 (pid : p_id_t) (i : input_t) (ss : share_t list) : output_t =
    let tl = snd i in
    let ss = correct ss in
    let pre = map2 (fun (sh : share_t) (x : t) => ScalarMultiplicationProtocol.pexec (sh, x)) ss tl
         in
    foldr (fun (x : share_t) (acc : share_t) =>
           if Pedersen_comp.PedersenCommitmentScheme.PedersenCS.verify (pid, fst x) (snd x) then
             AdditionProtocol.pexec (x,acc)
           else acc) (F.zero, (F.zero, [])) ss.
  op prot (rs : rands_t) (inps : inputs_t) : (convs_t * outputs_t) option =
    let pst1 = map (fun pid => (pid, pstage1 (oget (assoc inps pid)))) p_id_set in
    let cs = map (fun pid => (pid, map (fun ss => snd ss) (oget (assoc pst1 pid)))) p_id_set in
    let pst2 = map (fun pid => (pid, pstage2 (oget (assoc rs pid)) (oget (assoc pst1 pid))))
        p_id_set in
    let sss = map (fun pid => (pid, map (fun idss => oget (assoc (snd idss) pid)) pst2)) p_id_set
        in
    let os = map (fun pid => (pid, pstage3 pid (oget (assoc inps pid))(oget (assoc sss pid))))
        p_id_set in
    let cs = map (fun pid => (pid, oget (assoc cs pid) ++ (oget (assoc sss pid)))) p_id_set in
    Some (cs, os).
  clone import Protocol as MalMultiplication with
    ...
end MALMultiplicationProtocol.
```

Fig. 43: Malicious mul protocol