Machine-checked ZKP for NP-relations: Formally Verified Security Proofs and Implementations of MPC-in-the-Head

José Bacelar Almeida[1], Manuel Barbosa[2], Manuel L Correia[2], Karim Eldefrawy[3], Stéphane Graham-Lengrand[3], Hugo Pacheco[2], and Vitor Pereira[3]

[1]University of Minho and INESC TEC
[2]University of Porto (FCUP) and INESC TEC
[3]SRI International

Abstract

MPC-in-the-Head (MitH) is a general framework that enables constructing efficient zero-knowledge (ZK) protocols for NP relations from secure multiparty computation (MPC) protocols. In this paper we present the first machine-checked implementations of this transformation. We begin with an EasyCrypt formalization that preserves modular structure of the original MitH construction and can be instantiated with arbitrary MPC protocols, secret sharing and commitment schemes satisfying standard notions of security. We then formalize various suitable components, which we use to obtain full-fledged ZK protocols for general relations. We compare two approaches for obtaining verified executable implementations. The first approach realizes a fully automated extraction from EasyCrypt to OCaml. The second one reduces the trusted computing base (TCB) and provides better performance for the extracted executable by combining code extraction with manual formal verification of low-level components implemented in the Jasmin language.

We conclude the paper with a discussion of the trade-off between formal verification effort and performance, and also discuss how our approach opens the way for fully verified implementations of state-of the-art optimized protocols based on MitH.

Keywords: Zero-Knowledge; Secure Multiparty Computation; Formal Verification; Implementation

A shorter version of this paper will appear in the Proceedings of the 2021 ACM Conference on Computer and Communications Security (CCS 2021). This is the full version. This material is based upon work supported by DARPA under Contract No. HR001120C0086. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or DARPA. José Bacelar Almeida has been partially supported by the Portuguese Foundation for Science and Technology (FCT), project REASSURE (PTDC/EEI-COM/28550/2017), co-financed by the European Regional Development Fund (FEDER), through the North Regional Operational Program (NORTE 2020).

Contents

1 Introduction

The MPC-in-the-Head (MitH) paradigm was introduced by Ishai, Kushilevitz, Ostrovsky and Sahai [25] (IKOS) as a new foundational bridge between secure multi-party computation (MPC) and zero-knowledge proof (ZK) protocols. A ZK protocol for an NP relation[1] $R(x, w)$ can be seen as a two party computation where a prover $\mathcal{P}$ with input (x, w) and a verifier $\mathcal{V}$ with input x jointly compute the boolean function $f(x, w)$ that accepts the proof if and only if $R(x, w)$ holds. The MitH paradigm shows that there exists an *efficiency* advantage in considering MPC protocols for $n > 2$ and using a commit-challenge-response transformation to obtain a ZK protocol. The efficiency gain of MitH stems from two important observations: 1) that π only needs to satisfy a weak notion of security that allows for extremely efficient instantiations and, 2) that the round complexity of π has no impact in the final protocol, since π is evaluated "in-the-head". A series of follow-up works [26, 20, 16, 10, 5, 13, 9, 18, 22] demonstrated the efficiency and flexibility of protocol families inspired in the MitH core ideas, by exploring adaptations of this principle to specific MPC protocols and different ZK protocol (e.g., considering MPC protocols with preprocessing and different trust models, and ZK protocols with additional rounds). One notable takeaway of these works is that MitH allows for instantiations that can efficiently handle relations expressed as either arithmetic or boolean circuits. Moreover, the MitH paradigm has been used to create a new generation of post-quantum secure signatures such as Picnic, a notable candidate to the NIST post-quantum competition.[2]

In this work we explore the elegant simplicity and modularity of the MitH paradigm to obtain the an end-to-end machine-checked development, including security proofs and formally-verified implementations, for ZK protocols supporting general relations. We focus on the MitH variant that can be instantiated with passively secure secret-sharing-based MPC protocols that tolerate two corrupt (i.e., semi-honest) parties. This allowed us to build on an existing development that already provides a suitable instantiation for the underlying MPC protocol, secret sharing and commitment schemes.

Our work opens the way for formally verified implementations of more recent and optimized applications of the MitH paradigm. To illustrate this, we show that our development is flexible enough to allow for another instantiation based on (the passively secure variant of) the secret-sharing based protocol given by Maurer [27], for which we also give an optimized and formally verified implementation. Our optimizations cover both simplifications of the MitH construction and integration of verified high-speed assembly code. We further discuss the implications of our work at the end of this section. In more detail, our contributions are as follows:

- We formally specify and verify the foundational IKOS result [25] and its potential instantiations in EasyCrypt. The formalization is modular and relies on standard components, such as a secret sharing scheme, an MPC protocol, and a commitment scheme.

- We present two verified implementations resulting from two different MPC protocols. The first is an extension of the formally-verified implementation of the BGW protocol given in [19], which is fully automatically extracted to an OCaml implementation. The second one is a new machine-checked EasyCrypt formalization of the protocol by Maurer [27], for which we obtain a verified implementation by combining OCaml code extraction with verified high-speed assembly developed using the Jasmin framework [2]. This simultaneously reduces the trusted computing base (TCB) and improves performance.

[1] Intuitively: R is efficiently computable and s.t. solutions, if they exist, are short.
[2] https://microsoft.github.io/Picnic/

- As the main innovations of our work we highlight: 1) that this is the first end-to-end machine-checked implementation of a ZK protocol for general relations. 2) that the security proofs include the first formalization of foundational results in Zero Knowledge, such as reduction of soundness error by sequential composition and simulation by rejection sampling; and 3) that for the first time we explore the integration of verified assembly generated using the Jasmin framework with synthesized verified code obtained through code extraction.

Limitations. The formal proofs of security and correctness for the various constructions are complete, but the functional correctness proof for the Jasmin implementation covers only the addition and multiplication gates; it is currently being extended to the other gates. The TCB for the implementation based on BGW includes EasyCrypt and the extraction mechanism, as well as unverified OCaml libraries for multi-precision integers and cryptographic operations. For the implementation based on Maurer's protocol the TCB includes EasyCrypt (we rely less on the extraction mechanism), unverified OCaml libraries for basic data structures, and a thin layer of unverified hand-written C code connecting the extracted OCaml code to the Jasmin code. This layer handles memory allocation and randomness generation (not existing in Jasmin), as well as C wrappers for the assembly generated by the Jasmin compiler. In contrast to the first implementation, all the low-level cryptographic code is verified.

ZKP and MitH in practice ZK proofs are a critical component in many cryptographic protocols. There are two approaches to building them: i. fix a small class of relations (e.g., proving knowledge of a discrete logarithm) and develop a customized protocol for that setting (e.g., use Schnorr's protocol); or ii. define a general protocol that can be used to prove the validity of *any* relation, typically by proving satisfiability in a circuit model of computation. Protocols developed according to (i) are usually more efficient for the target family of statements; however, the second approach permits, not only handling arbitrarily complex statements, but also to *change* the statement that is being proved without redesigning the entire proof system from scratch.

MitH permits proving in ZK any relation expressed as satisfiability of an arithmetic circuit over a finite field. This is a functionally complete model of computation, i.e., it can be used to compute any Boolean function and decide any language in NP in the non-uniform complexity sense. Moreover, MitH admits post-quantum-secure instantiations [16], which is a significant advantage over alternative solutions that rely, for example, on bilinear pairings.

For concreteness we give here an example of where these protocols can be useful in practice. In the crypto-currency world (and block-chain in general) ZK proofs can be used to enable anonymous transactions. ZCash, for example uses ZK protocols to ensure that the information stored in the blockchain does not reveal where funds came from and where they went to. In this example, the *witness* is the private information that the sender (the prover) hides from the blockchain, including any signature keys required to sign the transaction. The verifiers will be the block-chain nodes consolidating the transaction and anyone checking the integrity of the blockchain. The statement/relation is specified by a boolean function that verifies the public information stored in the blockchain with respect to the private information withheld by the prover. Finally, the zero-knowledge property guarantees sender privacy, whereas the soundness property prevents the sender from cheating.

In practice there is a software engineering aspect to deploying such proofs that is also an interesting area of research. We note that the relation in this example is a complex computation, which may change when the protocol specification is modified. By using a protocol that supports general relation this should only require modifying the *circuit* that defines the statement and not the

entire protocol implementation. The work in this paper shows that high-assurance implementations of such protocols are within reach of the current formal verification technology.

Implications The MitH paradigm has received a lot of attention since the publication of the seminal IKOS paper [25], which we formalize in this paper. A natural question to ask is therefore: why not consider more recent and more efficient protocols? Our choice was motivated by 1) the goal of formally replicating the modular structure of the original IKOS construction; 2) a pragmatic approach to build on an existing development of the BGW protocol, which fits the original (foundational) view of [25]; and 3) the fact that the proofs of the soundness and (malicious verifier) zero-knowledge properties formalized in [25] posed an interesting challenge for machine-checking in EasyCrypt. Nevertheless, we believe that the complexity of the protocols, implementations and proofs we give here are representative of the challenges posed by more recent applications of the MitH paradigm.

This is immediate for the part of our work that focuses on verified implementations: code extraction and connection to verified Jasmin implementations can be done in essentially the same way, with the caveat that additional (non-cryptographic) verification effort is required to deal with implementation-specific optimizations such as compact view representations, parallel processing, etc.

Security proofs would require specific execution models that go beyond our syntax for MPC and ZK protocols (e.g., to capture preprocessing, probabilistic-checkable proofs, etc.), but we do not anticipate difficulties in formalizing proofs that can be expressed as standard game hopping arguments—this includes special soundness and honest-verifier zero-knowledge, which are also tackled in [30], and malicious security for MPC, which are tackled in [23, 19]. Two exceptions are the general Fiat-Shamir transformation and post-quantum security proofs (e.g., those relying on the QROM), which we believe are very interesting directions for future work.

Indeed as pointed out in [30], when it comes to EasyCrypt formalization, many recent protocols [26, 16, 9, 17, 29] fall into the same general category as the ZKBoo formalization given there (and indeed into our own). For example, to handle the ZKBoo [20] security proof similarly to what was done in [30], we would need to redefine views as intermediate states instead of messages exchanged — our intermediate passive security notions for MPC would remain the same — and adapt the notion of consistency in the same way. Furthermore, additional standard game hops would be needed to deal with the use of a PRF to compress randomness. When moving to witness-independent preprocessing as in [26], we would need to modify the syntax of our ZK protocols to deal with 5-rounds and formalize a simple cut-and-choose argument.

Access to the development. Our EasyCrypt formalizations, proofs, and extracted executable software are available in the following repository https://github.com/SRI-CSL/high-assurance-crypto/tree/main/high-assurance-zk.

2 Preliminaries

We provide in this section the formal cryptographic definitions that we use in our formalization, which are all standard, and an explanation of the MitH transform. We follow [25] closely and conclude the section with a short overview of the EasyCrypt and Jasmin features relevant to our work.

3 Basic Primitives

We note that, as the MPC protocols and secret sharing protocols we use to instantiate the MitH construction are information theoretically secure, the security of the commitment scheme determines the class of adversaries over the security of the MitH construction. In particular, the standard soundness property only holds if the commitment is statistically binding; if the commitment scheme is only computationally binding, then we obtain a ZK argument.

3.1 Secret Sharing

For a secret space W, which we will take to be the same as a share space, a secret sharing scheme for n parties is defined by two ppt algorithms:

- The probabilistic sharing algorithm $\mathsf{share}(w)$ takes a secret $w \in W$ and returns a sharing $\bar{w}$.
- The deterministic reconstruction algorithm $\mathsf{unshare}(\bar{w})$ takes a sharing $\bar{w}$ and returns a secret $w \in W$.

A secret sharing scheme should satisfy the following properties:

- Correctness: For all secrets $w \in W$ and all sharings $\bar{w} \in [\mathsf{share}(w)]$, we have that $w = \mathsf{unshare}(\bar{w})$.
- t-Privacy Any subset of t shares, for $0 \leq t < n$ can be simulated efficiently by sampling from a fixed public distribution.

3.2 Commitment Scheme

A commitment scheme is defined by two ppt algorithms:

- The probabilistic commitment algorithm $\mathsf{commit}(m)$ takes a message m and returns a pair (c, k) where c is a commitment and k is an opening.
- The deterministic verification algorithm $\mathsf{verify}(m, c, k)$ takes a message m, a commitment c, and an opening k and it returns 1 or 0 indicating success or failure, respectively.

A commitment scheme should satisfy the following properties:

- Correctness: For all messages m, commitments c and openings k, such that $(c, k) \in [\mathsf{commit}(m)]$, we have $\mathsf{verify}(m, c, k) = 1$.
- Binding: The commitment scheme is computationally (resp. statistically) binding if the probability that the following game returns 1 is bounded by a small ϵ when $\mathcal{A}$ is a ppt (resp. a potentially unbounded) algorithm:
  - the binding game first runs an adversary $\mathcal{A}$ who outputs a tuple (m, k, m', k', c);
  - the game then runs $\mathsf{verify}(m, c, k)$ and $\mathsf{verify}(m', c, k')$.
  - the game returns 1 if and only if both verifications are successful and $m \neq m'$.
- Hiding: The commitment scheme is computationally (resp. statistically) hiding if the probability that an adversary $\mathcal{A}$ returns 1 when running in the following game changes by at most a small ϵ, when $\mathcal{A}$ is a ppt (resp. a potentially unbounded) algorithm, and b is either 0 or 1:
  - the hiding game runs adversary $\mathcal{A}$ that chooses two messages m_0, m_1;
  - the game then computes $(c, k) \twoheadleftarrow \mathsf{Com}(m_b)$, which it provides to $\mathcal{A}$;
  - eventually $\mathcal{A}$ terminates outputting a guess b'.

3.3 Zero-Knowledge

A NP relation $R(x, w)$ is an efficiently decidable and polynomially bounded binary relation, which we see as a boolean function. A ZK protocol for a NP relation $R(x, w)$ is defined by two probabilistic polynomial time (ppt) interactive algorithms, a prover $\mathcal{P}$ and a verifier $\mathcal{V}$: $\mathcal{P}$ takes a NP statement x and a witness w; $\mathcal{V}$ is only given the statement x. The prover and the verifier interact—in this paper we consider only three-pass commit-challenge-response protocols—until eventually the verifier outputs 1 or 0 indicating success or failure, respectively. The view of $\mathcal{V}$ is defined as its input x, its coin tosses and all the messages that it receives.

Definition 3.1 (Zero-knowledge proof)**.** *A protocol $(\mathcal{P}, \mathcal{V})$ is a zero-knowledge proof protocol for the relation R if it satisfies the following requirements:*

- Completeness*: In an honest execution, if $R(x, w) = 1$, then the verifier accepts with probability 1.*
- Soundness*: For every malicious and computationally unbounded prover $\mathcal{P}^*$, there is a negligible function $\epsilon(\cdot)$ such that, if $R(x, w) = 0$ for all $w \in \{0,1\}^{p(|x|)}$, then $\mathcal{P}^*$ can make $\mathcal{V}$ accept with probability at most $\epsilon(|x|)$.*
- Zero-Knowledge*: For any malicious ppt verifier $\mathcal{V}^*$, there exists a ppt simulator $\mathcal{S}^*$, such that the view of $\mathcal{V}^*$ when interacting with $\mathcal{P}$ on inputs (x, w) for which $R(x, w) = 1$, is computationally indistinguishable from the output of $\mathcal{S}^*$ on input x.*[3]

We do not consider the proof-of-knowledge (PoK) property, which imposes that a witness can be extracted from any successful prover. The PoK property is not proved in [25], so we leave it for future work. We will also consider zero-knowledge protocols that have a constant (non-negligible) soundness error ϵ, in which cases the soundness error will be specified. In this case, the soundness error can be reduced to match the definition above by repeating the protocol multiple times, as discussed in Section 4.4.

3.4 Secure Multiparty Computation

The MitH paradigm builds on MPC protocols that assume synchronous communication over secure point-to-point channels. Let n be the number of parties, which will be denoted by $P_1, \ldots, P_n$. All players share a public input x, and each player P_i holds a local private input w_i. We consider protocols that can securely compute a n-input function f that maps the inputs $((x, w_1), \ldots, (x, w_n))$ to a n-tuple of boolean outputs. We will consider f to be a boolean function, so the output values given to all parties by f are the same.

A protocol Π is specified via its next-message function. That is, $\Pi(i, x, w_i, r_i, (m_1, \ldots, m_j))$ returns the set of n messages sent by P_i in round $j+1$, given the public input x, its local input w_i, its random coins r_i, and the messages $m_1, \ldots, m_j$ it received in the first j rounds. The output of the next message function Π may also indicate that the protocol should terminate, in which case Π returns the local output of P_i. The view of P_i, denoted by V_i, includes x, w_i, r_i and the messages received by P_i during the execution of Π. Note that Π and V_i fully define the set of messages sent by P_i and also its output. The following notions of consistency are important for the MitH transformation.

[3] Two distributions are indistinguishable if, for all distinguishers returning a bit, the probability that the distinguisher returns 1 when fed with a value sampled from either of the distributions changes by a small quantity ϵ. Our indistinguishability proofs are given as reductions, so they imply computational/statistical/perfect security when the underlying components are themselves computationally/statistically/perfectly secure.

Definition 3.2 (Consistent views)**.** *A pair of views V_i, V_j are consistent (wrt protocol Π and some public input x) if the outgoing messages implicit in V_i sent from party i to party j are identical to the incoming messages to j from i reported in V_j, and vice versa.*

Lemma 3.1 (Local vs. global consistency [25])**.** *Let Π be a n-party protocol and x be a public input. Let $(V_1, \ldots, V_n)$ be a n-tuple of (possibly incorrect) views. Then all pairs of views V_i, V_j are consistent with respect to Π and x if and only if there exists an honest execution of Π with public input x (and some choice of private inputs w_i and random inputs r_i) in which V_i is the view of P_i for every $1 \leq i \leq n$.*

We consider security of protocols in the semi-honest model. Correctness and privacy are defined in the standard way: in an honest execution the parties obtain the correct result, and t-privacy requires the existence of a simulator that can replicate the views of t corrupt parties without knowing anything about the honest parties' inputs.

Definition 3.3 (Correctness)**.** *Protocol π securely computes function $f((x, w_1), \ldots, (x, w_n))$ with perfect correctness if, for all inputs $x, w_1, \ldots, w_n$, the probability that the output of some player is different from the output of f is 0, where the probability is over the independent choices of the random inputs $r_1, \ldots, r_n$.*

Definition 3.4 (t-Privacy)**.** *Protocol π computes f with t-privacy, for $1 \leq t \leq n$, if there is a ppt simulator $\mathcal{S}$ such that, for any inputs $x, w_1, \ldots, w_n$ and every set of corrupted parties $T \subseteq [n]$, where $|T| \leq t$, the joint view $(V_{i_1}, \ldots, V_{i_k})$ of the parties in $T = \{i_1, \ldots, i_k\}$ is indistinguishable from $\mathcal{S}(T, x, [w_i]_{i \in T}, [f(x, w_1, \ldots, w_n)_i]_{i \in T})$.*

3.5 MPC-in-the-Head

We give here a view of the MitH construction that closely follows our formalization. We rely on MPC protocols where inputs to parties are encoded as a fixed number of elements in a finite field $\mathbb{F}_q$, for $q \geq 2$ prime: x encodes public information known about the statement to be proved; each w_i corresponds to a secret share of w, the witness known only to the prover. Here, w is itself a fixed, say k, number of elements in $\mathbb{F}_q$.

We write $(w_1, \ldots, w_n) \twoheadleftarrow \mathsf{share}(w)$ to denote the secret sharing operation and $w \leftarrow \mathsf{unshare}(w_1, \ldots, w_n)$ for unsharing, where the former is probabilistic. We set this secret sharing operation to the trivial splitting into n shares, where each w_i is a k-tuple in $\mathbb{F}_q$, the first $n-1$ shares are chosen uniformly at random, and w_n is computed as $w - \sum_1^{n-1} w_i$ with addition performed pointwise over $\mathbb{F}_q^k$. The crucial properties are perfect correctness (i.e., unsharing always recovers the witness) and 2-privacy (any two shares look perfectly random and reveal nothing about w).

MPC computations are specified by algebraic circuits over $\mathbb{F}_q$, i.e., sequences of additions and multiplications over values in $\mathbb{F}_q$. Note that this computational model is functionally complete. We start by fixing an arbitrary circuit C that computes a boolean function $f(x, w_1, \ldots, w_n)$, and an MPC protocol that guarantees C is computed correctly and securely as above. We will impose that f is such that its output depends only on the value that results from unsharing $(w_1, \ldots, w_n)$, i.e, it must hold that $f(x, w_1, \ldots, w_n) = f(x, w'_1, \ldots, w'_n)$ if $\mathsf{unshare}(w_1, \ldots, w_n) = \mathsf{unshare}(w'_1, \ldots, w'_n)$. The MitH construction then yields a ZK protocol that permits proving statements of the form

$$R(x, w) := \exists (w_1, \ldots, w_n), \begin{cases} (w_1, \ldots, w_n) \in \mathsf{share}(w) \\ f(x, w_1, \ldots, w_n) = 1 \end{cases} . \tag{1}$$

This is essentially the same as defining function f on all points according to $f(x, w_1, \ldots, w_n) := R(x, \mathsf{SS.unshare}(w_1, \ldots, w_n))$, and then building a circuit C for it. In our work we do not handle

the constructive step of building a circuit C, and we formalize instead the relation induced by a well-formed circuit.[4]

In short, the MitH ZK protocol runs in three steps as follows:

Commit The prover on input (x, w) first takes the witness w and secret shares it into $(w_1, \ldots, w_n)$. It then executes the full MPC protocol (emulating all parties in the head) on inputs $(x, w_1, \ldots, w_n)$. The prover creates n commitments that bind it to the views of the n parties (i.e., their inputs, the randomness that they used, and the messages they received); the commitments are sent to the verifier.

Challenge The verifier chooses a pair of parties (i, j) uniformly at random and sends this challenge to the verifier.

Response The prover sends the views of parties (i, j) to the verifier by opening the corresponding commitments.

Verification The verifier checks that the views of the parties it received satisfy the following properties: (i) they are consistent with the originally received commitments (ii) they are mutually consistent wrt the messages sent and received (sent messages can be recomputed using the randomness and incoming messages so far by each party), and (iii) both parties conclude outputting 1. The verifier accepts if and only if all checks are completed successfully.

We discuss the proofs of the completeness, zero knowledge and soundness properties of this construction in the next section, when we explain our formalization. We now give a short intuition.

Completeness simply means that, if both entities are honest, then the protocol completes successfully; this follows from the correctness of all components.

The zero knowledge property imposes that the verifier cannot extract useful information about the witness. This is guaranteed because all the verifier sees are the views and input shares of two parties, and these do not reveal any information about the witness down to the 2-privacy properties of the secret sharing scheme and the MPC protocol, as well as the hiding property of the commitment scheme.

Finally, soundness is the less intuitive property. One must first observe that, to cheat, the prover must find a secret sharing of some witness, such that the parties in the MPC protocol viewed by the verifier return 1 and, furthermore, that the views of these parties are consistent with each other. If the prover is cheating, then no witness exists such that this is true for all pairs of parties (this follows from Lemma 3.1). This means that the prover can only cheat if the verifier's challenge happens to miss a pair of inconsistent views; since there exists at least one such pair, the verifier will catch the prover with probability at least $1/\binom{n}{2}$. Note that choosing the smallest possible n in this case minimizes the soundness error, which is why we instantiate the MPC protocols with $n = 5$: the smallest n such that 2-privacy is guaranteed to hold. Other variants of MitH permit setting different tradeoffs.

3.6 Background on EasyCrypt and Jasmin

EasyCrypt is an interactive proof assistant tailored for cryptography. EasyCrypt adopts the code-based approach, in which primitives, security goals and hardness assumptions are expressed as probabilistic programs. It supports modular reasoning via its theory system, where a theory is a

[4]Note that the relation induced by f in Equation 1 is well-defined even when share is not surjective over the space of shares; this is not relevant for the trivial splitting we described above, but it is relevant for the optimized instantiation of MitH using Maurer's protocol that we describe in Section 5.

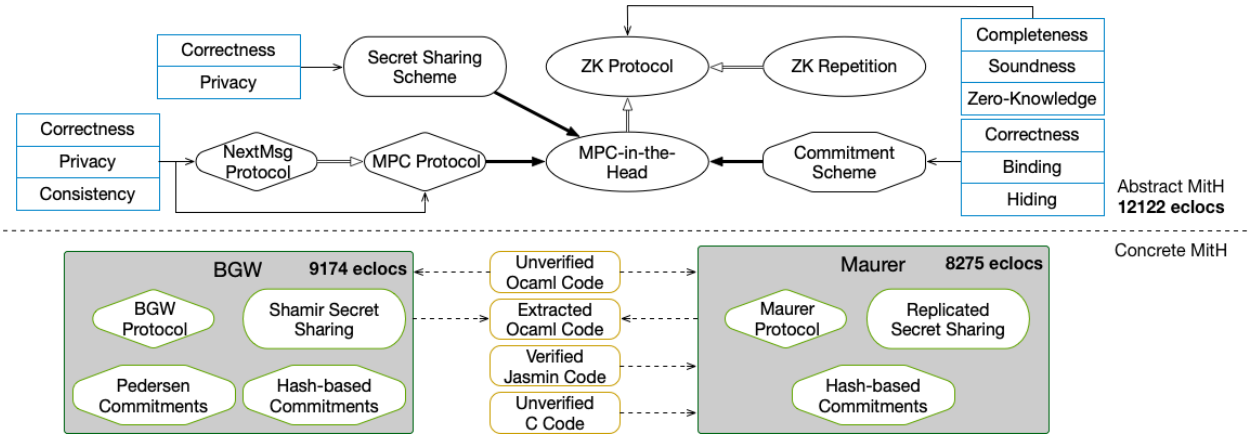

Figure 1: Overview of the relation between different parts of our formalization, annotated with overall lines of proof code. The horizontal dotted line separates abstract MitH components (above) from concrete instantiations (below). Black and green shapes represent abstract cryptographic constructions and their respective instantiations. Blue rectangles depict security definitions applied to the connected cryptographic constructions. Double arrows and bold arrows indicate component instantiation and sub-components, respectively. Yellow rectangles denote executable code for the implementations.

collection of types and operators, and it also provides a cloning mechanism that allows theories to refine the types and operators of more abstract theories, or to combine the elements of existing ones.

The EasyCrypt code extraction tool used in this paper has been recently presented in [19]. This tool allows to extract OCaml executable code from an EasyCrypt proof script where code is defined via functional operators. We note that code extraction to OCaml is a common way to obtain high-assurance code that is correct by construction and it is used in the Jasmin compiler itself and other formally verified tools such as CompCert.

Jasmin [2] is a pre-assembly language that was developed for high-speed and high-assurance cryptography. Jasmin implementations are predictably transformed into assembly programs by the Jasmin compiler, which is formally verified in Coq. Predictability empowers Jasmin programmers to develop optimized implementations with essentially the same level of control as if they were using assembly. Jasmin source-code can also be extracted into an EasyCrypt representation, supported by an axiomatic semantics. Taken together, Jasmin and EasyCrypt provide a convenient framework to develop efficient verified implementations.

4 Machine-checked MPC-in-the-Head

Our EasyCrypt development defines an abstract and modular infrastructure that follows the general MitH transformation, and can be instantiated with different concrete components. Figure 1 depicts the relation between these different components in our formalization. In what follows, we will provide a more detailed view, resorting directly to snippets of EasyCrypt code simplified for readability. In subsequent sections we discuss concrete instantiations and implementations.

4.1 ZK Protocols and MitH Building Blocks

The following definitions fix the commit-challenge-response three pass protocol structure of ZK protocols in EasyCrypt, since this is all that is required for the MitH transformation. These are the types and operators that must be defined by concrete ZK protocols.

```
op relation : witness_t → statement_t → bool.
op language(x : statement_t) = ∃ w, relation w x = true.

type prover_input_t = witness_t * statement_t.
type verifier_input_t = statement_t.
type prover_output_t = unit.
type verifier_output_t = bool.

op commit : prover_rand_t → prover_input_t →
                                        prover_state_t * commitment_t.
op challenge : verifier_rand_t → verifier_input_t → commitment_t →
                                        verifier_state_t * challenge_t.
op response : prover_state_t → challenge_t → response_t.
op check : verifier_state_t → response_t → bool.

type trace_t = commitment_t * challenge_t * response_t.

op protocol (r : prover_rand_t * verifier_rand_t)
            (x : prover_input_t * verifier_input_t)
            : trace_t * (prover_output_t * verifier_output_t) =
  let (r_p, r_v) = r in let (x_p, x_v) = x in
  let (st_p,c) = commit r_p x_p in
  let (st_v,ch) = challenge r_v x_v c in
  let r = response st_p ch in
  let b = check st_v r in ((c, ch, r), ((),b)).
```

Types that are undefined at this level must be specified by each protocol. This is the case, for example, for the types of witnesses and statements, but not for the outputs of the prover and verifier, which are hardwired in the syntax to be the singleton type and a boolean value, respectively. Each protocol is associated with a relation, which at this level is modeled as an abstract boolean function. Finally, the theory also defines what it means to honestly execute the protocol via the `protocol` operator. Note that all algorithms are derandomized, in the sense that they take randomness sampled from elsewhere. This is because in our implementations we also must follow this structure, and our results assume that randomness used by honest parties is sampled uniformly at random both at the specification and implementation levels.

All the security properties are defined as EasyCrypt games. These games are parameterized by adversarial entities, over which the definitions of security are quantified (e.g., malicious provers/verifiers) and also by modules that capture the ideal sampling of randomness for the honest parties. As a simple example, the completeness property is defined by the following game

```
module Completeness(R : Rand_t) = {
  proc main(w : witness_t, x : statement_t) : bool = {
    (r_p, r_v) <@ R.gen();
    (tr,y) ← protocol (r_p, r_v) ((w,x), x);
    return (snd y); } }.
```

The game is defined within a *module*, which is parameterized by another module `R` that samples randomness. This is because randomness sampling procedures must be specified for each protocol. The experiment calls the procedure `R.gen` (special syntax `<@` is used for procedure calls) to obtain randomness for both prover and verifier, runs the full protocol, and outputs the result produced by the verifier. A completeness claim in EasyCrypt can be written as:

```
∀ w x, relation w x ⇒ Pr [ Completeness(R).main(w,x) : res ] = 1.
```

Here *res* is a reserved word in EasyCrypt that refers to the event that a procedure returning a boolean value outputs *true*.

Soundness. For the soundness definition, we need to quantify over potentially malicious provers. In EasyCrypt this is done by defining a module type, i.e., the interface that the adversary exposes. Module type `MProver_t` specifies this interface.

```
module type MProver_t = {
  proc commitment (x: statement_t) : commitment_t
  proc response(x : statement_t, c : commitment_t, ch : challenge_t) : response_t}.
```

Observe that a malicious prover keeps arbitrary internal state and can sample arbitrary randomness which is out of control of the security experiment. The soundness property can now be expressed as a game parameterized by an attacker of this type, in addition to the module that samples randomness for the honest verifier. This allows us to quantify universally over malicious provers.

```
module Soundness(RV : RandV_t, MP : MProver_t) = {
  proc main(x : statement_t) : bool = {
    r_v <@ RV.gen();
    c <@ MP.commitment(x);
    (st_r, ch) ← challenge r_v x c;
    resp <@ MP.response(x, c, ch);
    return (check st_r resp); } }.
```

A soundness claim, for some fixed `RV`, can be written as:

```
∀ x MP, !language x ⇒ Pr [ Soundness(RV,MP).main(x) : res ] ≤ epsilon.
```

Zero Knowledge. We formalize two versions of the zero knowledge property. We present here the one that we use to obtain a first (intermediate) result for the concrete protocol produced by the MitH construction (we call this the single-run zero-knowledge property). We defer to Section 4.4 an explanation of how this property then allows us to derive a proof for the standard zero-knowledge property using a repetition argument, as in [25].

The single-run zero knowledge property is formalized by defining two experiments, typically known as the *real* and *ideal* worlds. We capture both worlds with a single module `ZKGame`, which can be parameterized by a real-world evaluator or an ideal-world evaluator.

```
module type MVerifier_t = {
  proc challenge(x : statement_t, c : commitment_t) :
                                          challenge_t * verifier_state_t }.

module type Evaluator_t (MV : MVerifier_t) = {
  proc eval(w : witness_t, x : statement_t, rp : prover_rand_t) :
                                      (verifier_state_t * trace_t) option }.

module type Distinguisher_t = {
  proc guess(tr : witness_t * statement_t *
                              (verifier_state_t * trace_t) option) : bool }.

module type Simulator_t = {
  proc commitment(x : statement_t) : commitment_t option
  proc response(x : statement_t, ch : challenge_t) : response_t option }.

module ZKGame (D : Distinguisher_t) (RP : RandP_t)
                     (E : Evaluator_t) (MV : MVerifier_t) = {
  proc main(w : witness_t, x : statement_t) : bool = {
    rp <@ RP.gen(); ctr <@ E(MV).eval(w,x,rp); b <@ D.guess(w,x,ctr);
    return b; } }.
```

Such an evaluator either outputs a protocol execution trace or a failure symbol (represented by *option*). This trace is given to a distinguisher, which produces a bit. Intuitively, if the two worlds are indistinguishable, then the distinguisher will output 1 with essentially the same probability in either one. The `Real` evaluator module animates the interaction of a malicious verifier `MV` (with type shown above) with the prover.

```
module (Real : Evaluator_t) (MV : MVerifier_t) = {
  proc eval(w : witness_t, x : statement_t, rp : prover_rand_t) :
                              (verifier_state_t * trace_t) option = {
    cp <$ chald; r ← None;
    (stp,c) ← commit rp (w,x);
    (ch,vst) <@ MV.challenge(x, c);
    if (ch = cp) { resp ← response stp ch; r ← Some (vst,(c,ch,resp)); }
    return r; } }.
```

In the standard zero knowledge property this module would simply output the execution trace. However, in the single-run ZK property, the real-world experiment further samples a challenge uniformly at random from the set of challenges (denoted `cp <$ chald`) and, before returning the execution trace, it checks whether the challenge occurring in the protocol (chosen by the malicious prover) matches the independently sampled one; it outputs the trace if this is the case or a failure symbol otherwise.

This intuitively means that, in our single-run zero-knowledge property, the simulator in the ideal world will only need to match the real-world execution trace when it guesses the challenge produced by the verifier.

The `Ideal` evaluator module, which we omit due to space constraints, animates a unique interaction between a simulator and the malicious prover. The goal of the simulator is to present to the verifier a view that is indistinguishable from a real execution, without knowing the witness. Note that the simulator in this definition is very limited, since in the general definition of the ZK property the simulator is given a description of the malicious verifier, which it may run an arbitrary number of times. For this reason, unlike an honest prover, we allow the simulator to signal an abort, in which case the ideal world evaluator will also return a failure symbol. This may happen if the simulator's strategy does not always work, for example because it guesses ahead of time what the malicious verifier will be doing.

A single-run zero-knowledge claim, for a concrete simulator `S` and randomness generation model `RP`, can be written as:

```
∀ w x D MV, relation w x ⇒
   | Pr [ ZKGame(D,RP,Real(MV)).main(w,x) : res ] -
     Pr [ ZKGame(D,RP,Ideal(S,MV)).main(w,x) : res ] | ≤ epsilon.
```

4.2 MPC Protocols

Our formalization of MPC protocols shares many similarities with the formalization of ZK protocols given above; it is the same as that used in [19] The main difference, in addition to considering an arbitrary number of parties, is that we need to consider protocols that are parametric on an abstract type for circuits (i.e., a way to represent n-input to n-output computations) and an abstract operator that defines what it means to evaluate an arbitrary circuit.

```
type circuit_t.

type input_t = pinput_t * sinput_t.
op f : circuit_t → (pid_t * input_t) list → (pid_t * output_t) list

type view_t = input_t * rand_t * in_messages_t.
type views_t = (pid_t * view_t) list.
```

```
op protocol : circuit_t → (pid_t * rand_t) list → (pid_t * input_t) list
                → (pid_t * in_messages_t) list * (pid_t * output_t) list.
```

Party inputs `input_t` can be defined as having a public and a secret part, which may model multiple input wires to the circuit. Indeed, it is the circuit evaluation operator `f` that will define the semantics of evaluating a circuit on given inputs. Finally, the `protocol` operator is used to define the global protocol evaluation, and it allows flexibility in defining the message scheduling, e.g., following the next-message function approach introduced in Section 2

Given these definitions we can capture the notion of pairwise view consistency as the following axiom corresponding to Lemma 3.1.

```
op consistent_trace c xp vs =
   (∀ (i j : pid_t), consistent_views c xp (assoc vs i) (assoc vs j) i j).

axiom local_global_consistency (c : circuit_t) xp (vs : views_t) :
  valid_circuit c ⇒ consistent_trace c xp vs ⇔ (∃ rs sx, valid_rands c rs ∧
    let xs = mk_inputs xp sx in let (tr,y) = protocol c rs xs in
     unzip1 sx = pid_set ∧ valid_inputs c xs ∧ (∀ pid, pid ∈ pid_set ⇒
      assoc vs pid = (assoc xs pid, assoc rs pid, assoc tr pid))).
```

Here, `mk_inputs` is a simple operator that constructs full inputs from public and secret inputs, and `assoc` is the operator that retrieves an element from an association list (in this case indexed by the party identifier). This axiom implies the existence of a `consistent_views` operator that can be used by the verifier in the MitH construction (under a few validity restrictions). This is the only required consistency property at this level of abstraction. For concrete instantiations the operator must be made concrete and proved to satisfy the assumption stated here as an axiom. We have formalized and proved this property for any protocol that follows the next-message syntax, and refined it to to a proof for our instantiation based on Maurer's protocol (cf. Section 5). Our BGW instantiation currently leaves this property as an axiom, but could be adapted in the same way to match the next-message syntax.

The correctness of MPC protocols is formalized analogously to completeness for ZK protocols. The t-privacy property is formalized using the same approach used for the zero-knowledge property of ZK protocols, using a real world and an ideal-world operation. The main difference is that the simulator must now construct t views to be fed to a distinguisher. We omit the games due to space constraints but give the simulator type here. Note that the simulator receives the party identities and full inputs for the corrupted parties, as well as their outputs, and must produce full views that reflect the interaction with honest parties without knowing their secret inputs. If such a simulator exists, then the secrets of honest parties are protected by the MPC protocol.

```
module type Simulator_t = {
  proc simulate(c : circuit_t, xs : (pid_t * (pinput_t * sinput_t)) list,
                    ys : (pid_t * output_t) list) : (pid_t * view_t) list }.
```

4.3 Formalizing the MitH Transformation

We follow the modular structure of [25] in our formalization, so we rely on the EasyCrypt theory cloning mechanism to obtain a formalization that is parametric on sub-theories for the MitH building blocks. We fix the number of parties to $n = 5$, as this allows us to explicitly unfold some of the hybrid arguments that appear in the proof and reduce proof complexity.

```
type witness_t.
type statement_t.

clone import SecretSharingScheme as SS with
```

```
  type secret_t = witness_t, op n = 5, op t = 2.

clone import Protocol as MPC with
  op n = SS.n, op t = SS.t,
  type pinput_t = statement_t, type sinput_t = share_t, type output_t = bool.

op relc : circuit_t.

clone import CommitmentScheme as CS with type msg_t = view_t.

axiom good_circuit (x : statement_t) w : valid_circuit relc ∧
  (∀ (ss ss' : (pid_t * sinput_t) list), unshare ss = w ⇒ unshare ss' = w ⇒
    let fss = f relc (mk_inputs x ss) in let fss' = f relc (mk_inputs x ss') in
    fss = fss' ∧ (∃ b, all (fun x ⇒ snd x = b) fss)).

op relation (w : witness_t) (x : statement_t) =
  ∃ (ss : (pid_t * share_t) list), w = unshare ss ∧ valid_share ss ∧
    let outs = f relc (mk_inputs x ss) in all (fun x ⇒ snd x) outs.
```

We start with abstract types for witnesses and statements. We then import the definitions for a secret sharing scheme, where secrets to be shared are of our witness type, we fix the number of parties to 5, and require $(t = 2)$-privacy. The secret sharing scheme fixes the type of secret shares. We can then import the definitions for an MPC protocol, fixing the type of public inputs for all parties to that of statements, and the type of secret inputs to a secret share. The outputs of all parties will be a boolean value. The MPC protocol fixes the type of circuits and the semantics for evaluating circuits via function `f`. It also fixes the types of party views, which we then use to refine a general theory for commitments.

The ZK protocol resulting from the transformation will be relative to an arbitrary circuit `relc` that will be known to both prover and verifier. We restrict our attention to a special class of circuits by using axiom `good_circuit`: we consider only well formed circuits whose output, for any two secret sharings that represent the same value, is the same for all parties. Finally, the relation holds for any pair (x, w) such that the circuit `relc` outputs *true*, when evaluated on a set of secret shares that are a valid sharing of `w` (`valid_share` means that `ss` is in the range of the sharing operation). Intuitively, it suffices that the prover is able to find a sharing of `w` such that the MPC circuit accepts for the relation to hold. Note that, apart from the refinements shown in the Figure, all other type definitions remain abstract and can be instantiated arbitrarily.

At this point it is possible to define the types of messages exchanged by the ZK protocol resulting from the transformation: the prover's commitment message is a list of commitments corresponding to the views of the five MPC protocol parties; the challenge returned by the verifier is a pair of party identifiers, and the response returned by the prover is a pair of opening strings for the selected views.

```
  type commitment_t = (pid_t * CS.commitment_t) list.
  type challenge_t = pid_t * pid_t.
  type response_t = (view_t * CS.opening_t) * (view_t * CS.opening_t).
```

We omit the details of the types of the randomness taken by prover and verifier. In the case of the prover this includes all the randomness required for secret sharing, emulating the MPC protocol and generating the commitments. The case of the verifier is simpler, since its randomness includes only the choice of party identifiers for the challenge.

We give here the commitment generation operator, i.e., the first stage of the prover in the MitH construction, where the prover emulates the MPC protocol execution and commits to the view of all parties.

```
  op gen_commitment (rp : prover_rand_t) (xp : prover_input_t) :
                                prover_state_t * commitment_t =
```

```
  let (w,x) = xp in
  let (r_ss, r_mpc, r_cs) = rp in

  let ws = SS.share r_ss w in

  let x_mpc = mk_inputs x ws in
  let (tr,y) = MPC.protocol relc r_mpc x_mpc in

  let vs = construct_views x_mpc r_mpc tr SS.pid_set in
  let cvs = map (fun pid ⇒
                let r_c = oget (assoc r_cs pid) in
                let v = oget (assoc vs pid) in
                (pid, (v, commit r_c v))) SS.pid_set in
  let cs = get_commitments cvs SS.pid_set in (cvs, cs).
```

The challenge and response steps are simpler to formalize. The challenge is simply a random sampling of a pair of party identifiers, which translates to copying random values from the randomness input. The response simply selects the views and opening strings corresponding to the selected parties, which are kept as internal state by the prover.

Finally the verifier checks the response as follows.

```
op check (xv : verifier_input_t) (cs : commitment_t)
            (rv : verifier_rand_t) (r : response_t) : bool =
  let (i,j) = rv in

  let (vosi, vosj) = r in let (vi, osi) = vosi in let (vj, osj) = vosj in

  let (xi,ri,tri) = vi in let (xj,rj,trj) = vj in

  let ci = get_party_commitment i cs in
  let cj = get_party_commitment j cs in

  CS.verify vi (ci,osi) ∧ CS.verify vj (cj,osj) ∧
  MPC.consistent_views relc xv vi vj i j ∧ MPC.valid_inputs xv vi vj i j ∧
  MPC.local_output relc i (xi,ri,tri) ∧ MPC.local_output relc j (xj,rj,trj).
```

The `check` operator verifies four conditions: 1) that the commitment openings are valid wrt to the provided views; 2) that the provided views are consistent with each other (using operator `consistent_views`); 3) that the inputs to the MPC protocol are well-formed; and 4) that the local output of the selected parties is *true*, which implies that the MPC protocol execution for these parties reported that the relation between statement and witness holds.[5]

Completeness Our completeness theorem states that the MitH construction has perfect completeness assuming perfect correctness for the underlying components. Formally, in EasyCrypt we prove that, for all valid randomness samplers `R`, all statements `x` and all witnesses `w`, the completeness experiment returns *true* with probability 1. The proof intuition is as follows. The `good_circuit` restriction imposes that the circuit that defines the relation is well behaved, in the sense that, for all sharings $\bar{w}, \bar{w}' \in [\mathsf{Share}(w)]$, the circuit outputs the same consistent values for all parties. Then, if the MPC protocol is correct, it will correctly compute the relation of the ZK proof system and every two views will be pairwise consistent (by Lemma 3.1). Since this is an honest execution, the commitments are well constructed and the openings will be valid, as per the correctness property of the commitment scheme.

[5]The check for well-formed MPC inputs in step (3) is trivially true when all possible share values of the input are in the range of the secret sharing scheme used by the prover, which is the case for the trivial additive splitting we described in Section 2.

Soundness The soundness theorem bounds the soundness error of the MitH construction by $1 - 1/\binom{n}{2} + \epsilon$ in a single execution. Here n is the number of parties in the MPC protocol and ϵ is bounded using the binding property of the commitment scheme. The proof is done by a sequence of two game hops. In the first hop we specify a bad event that checks if the dishonest prover opened a commitment for the first view requested by the verifier that is not the originally committed one; we then upper bound the probability of this event by writing an explicit reduction to the binding property of the commitment scheme. The second hop repeats the reduction, but for the second opened view. Finally, we prove that the verifier catches a cheating prover with probability at least $1/\binom{n}{2}$ as follows. Lemma 3.1 tells us that, either some party outputs *false*, in which case the verifier rejects, or there exists a pair of inconsistent views the set of views committed by the malicious prover. The bound follows because the honest verifier chooses a pair of views uniformly at random after the commitment pass.

Zero-knowledge For the Zero-Knowledge property, we must construct a simulator that deals with the fact that the malicious verifier may choose the challenge arbitrarily after seeing the first pass of the protocol. As mentioned above, we first prove that we can construct a good simulator for the single-run ZK definition, where the simulator only needs to work if it correctly guesses in advance the challenge that the malicious verifier will output. The reduction to the standard ZK property is proven in Section 4.4.

Our simulator therefore generates a challenge uniformly at random and runs the MPC simulator to generate the two views that will be opened to the malicious verifier. It fixes the remaining views to an arbitrary value. Note that we can fix the outputs of corrupt parties given to the MPC simulator to *true*, since this is the output of the computation when the MPC protocol is executed by an honest prover (indeed, for this proof, the MPC simulator only needs to work for executions where the outputs of the computation accept the pair (x, w)). Our single-run ZK simulator completes these views by sampling two random shares that will pose as the secret inputs for the corrupt MPC parties. Finally, it commits to all views to get a simulated first round message for the MitH construction. When computing the response, this strategy fails if the challenge guess was wrong. Otherwise it returns the simulated views.

The proof that this is a good single-run simulator uses a sequence of hops. In the first three hops, we replace the view of every party different from i and j, where (i, j) is the initially sampled (guessed) challenge, by an arbitrary value (we use the constant `witness` that is defined for all types in EasyCrypt). For each such hop, we can construct adversaries `B11`, `B12`, `B13` that break the hiding property of the commitment scheme whenever the distinguisher can detect the modification to the game. These adversaries choose to be challenged on either the real view or `witness`; they interpolate between the game in which a correct commitment is given to the verifier and the modified game by running the rest of the experiment themselves, providing the resulting view to the distinguisher, and outputting the distinguisher's guess. These three hops lead to a term in the final statement that is bounded by the added advantages of `B11`, `B12`, and `B13`, against the commitment scheme.

In the next step of the proof we replace the execution of the MPC protocol by an execution of the MPC simulator. Since, by assumption, the MPC protocol achieves 2-privacy, it is possible to bound the difference between the two games as the advantage of a distinguisher `B2` against the privacy of the MPC protocol: any change in the output distribution of the ZK experiment can be used to construct a distinguisher against the MPC simulator.

Finally, in the last hop, we replace the secret inputs given to the malicious MPC parties at the MPC simulator input with random shares (rather than shares generated from the true witness w). This hop follows from the t-privacy of the secret-sharing scheme, which we again prove by providing

an explicit reduction B3. The proof is completed by showing that this final game is identical to the ideal single-run ZK game when instantiated with our simulator.

The result proved in EasyCrypt is therefore of the form

```
lemma zero_knowledge w x D MV :
  | Pr [ ZKGame(D,RP,Real(MV)).game(w,x) : res ] -
    Pr [ ZKGame(D,RP,Ideal(MV,S(S_MPC))).game(w,x) : res ] | ≤ e1 + e2 + e3
```

where e1 sums the advantages of attackers B11, B12, B13, against the hiding property of the commitment scheme; e2 is the advantage of distinguisher B2 against the MPC simulator, and e3 is the advantage of attacker B3 against the privacy of the secret sharing scheme.

4.4 Meta Theorems

We have created a library of general results that are relevant, not only for the MitH transformation, but also for MPC protocols and Zero Knowledge protocols in general. The first part of the library deals with repetition arguments in Zero Knowledge proofs. The second part of the library formalizes general properties of MPC protocols that follow the next message syntax. The lemmas we prove can be instantiated with the concrete protocols we have developed, but we have not done so for the ones focusing on ZK; there is no technical impossibility in doing this, but it will imply a significant formalization and proof effort of boiler-plate equivalence proofs that express the behavior of concrete imperative algorithms in terms of functional operators.

Repetition in Zero Knowledge In this library we prove two general results, which imply that our proof of the MitH transformation as presented in the previous section actually implies the standard level of security for a ZK protocol. For the zero-knowledge property, we show that the single-run ZK proof implies the existence of a simulator that works for any malicious verifier with overwhelming probability. Intuitively, the full simulator repeats the full single-run ZK ideal evaluation until the single-run simulator succeeds. Since these are independent executions of a probabilistic experiment, this is essentially a rejection sampling of the simulated trace, which no distinguisher will be able to tell apart from a real execution (except if the single-run simulator always fails, which happens with negligible probability in the number of attempts). For the soundness property, we show that sequential composition of the protocol can be used to reduce the soundness error to an arbitrarily small value. In the future, the theory that contains these results will be extended to include a proof that the zero-knowledge property is also preserved by sequential composition. These are foundational results in cryptography which, to the best of our knowledge have not been formally specified and verified.

Our theory declares the types of provers and verifiers as follows.

```
type Prover_t = {
 commit: pauxdata_t → witness_t → statement_t → (commitment_t*pstate_t) distr;
 response: pstate_t → challenge_t → response_t * pauxdata_t }.

type Verifier_t = {
 challenge: vauxdata_t → statement_t → commitment_t → (challenge_t*vstate_t) distr;
 check: vstate_t → response_t → bool * vauxdata_t }.

module IPS = {
 (* prover/verifier auxiliary inputs *)
 var paux: pauxdata_t
 var vaux: vauxdata_t
 var fullview: fullview_t
 var view: view_t

 (* a single execution of the protocol *)
```

```
 proc exec(_P: Prover_t, _V: Verifier_t, _w: witness_t, _x: statement_t):bool={
  (com, pst) <$ _P.commit paux _w _x;
  (chlv, vst) <$ _V.challenge vaux _x com;
  (resp, paux) ← _P.response pst chlv;
  (b, vaux) ← _V.check vst resp;
  view ← (com,chlv,vst,resp);
  return b; }
 (* N-sequential repetitions of the protocol *)
 proc execN(_N:int,_P:Prover_t,_V:Verifier_t,_w:witness_t,_x:statement_t):bool={
  fullview ← []; b ← true; i ← 0;
  while (b && i < _N) { b <@ exec(_P,_V,_w,_x);
                        fullview ← view::fullview; i ← i + 1; }
  return b; } }.

(* a concrete Interactive Proof-System (P,V) *)
op P: Prover_t.
op V: Verifier_t.
```

The IPS module defines the execution environments for single execution (exec) and sequential composition (execN); both execution environments can be parameterized by honest or malicious provers/verifiers, depending on the property we are capturing. All algorithms are abstract operators in our proofs, but we fix an arbitrary zero-knowledge protocol by declaring global operators P and V. (Note the inclusion of auxiliary data in the syntax of provers and verifiers; this is not relevant when considering a single execution of the protocol, but it is critical for proving security under composition.) Completeness and soundness of sequential composition are then proved as the following EasyCrypt lemmas, where the stated axioms capture the hypotheses that completeness and soundness hold for a single execution of the protocol.

```
axiom completeness1 w x:
 R w x ⇒ Pr [ IPS.exec(P,V,w,x) : res ] = 1.

lemma completenessN N w x:
 0 < N ⇒ R w x ⇒ Pr [ IPS.execN(N,P,V,w,x) : res ] = 1.

op sound1_err : real.
axiom soundness1 w x P':
 (∀ w, R w x = false) ⇒ Pr [ IPS.exec(P',V,w,x) : res ] ≤ sound1_err.

lemma soundnessN N w x P': 0 < N ⇒
 (∀ w, R w x = false) ⇒ Pr [ IPS.execN(N,P',V,w,x) : res ] ≤ sound1_err^N.
```

The proofs of these lemmas are very similar to each other, as they use the while rule of EasyCrypt to derive the conclusion by induction; note that the argument *accumulates* a probabilistic event, so this is a very good illustrative example of the power of the probabilistic Hoare logic offered by EasyCrypt.

We now explain how we extend the single-run result we obtained for the MitH simulator to a full proof of the Zero-Knowledge property of the construction. This was one of the most challenging parts in our development, as the original proof is non-trivial and we needed to re-express it in terms that could be formalized in EasyCrypt. Recall what we have proved: we have a partial simulator that guesses the malicious verifier's challenge and, if the guess is correct, can produce an indistinguishable view of the protocol execution. What we set out to prove is that, given a concrete malicious verifier, we can construct a full simulator by attempting the partial simulation until eventually it succeeds, i.e., we perform up to N independent simulation attempts, and abort if none of them succeed. This simulator will be good enough if we make N sufficiently large.

We express our proof goal using the following module, which can be parameterized with the number of simulation attempts.

```
module ZK (D: DRoI_t) = {
```

```
proc simulator(_V: Verifier_t, _vaux: vauxdata_t, _x: statement_t)
                                                    : view_t*vauxdata_t = {
 i ← 0; bad ← true;
 while (bad && i < Nsim) {
  chl <$ rnd_challenge;
  (com, sst) <$ scommit _x chl; (chlv, vst) <$ _V.challenge _vaux _x com;
  bad ← !good_challenge chl chlv;
  if (!bad) {
   resp ← sresponse sst chlv;
   (b, vaux') ← _V.check vst resp;
   t ← (com,chlv,vst,resp); }
  i ← i + 1; }
 return (t,vaux'); } }.
```

We state as an assumption the single-run zero-knowledge result we proved in the previous section, by assuming that no distinguisher in the single-run experiment can change its behavior by a probability greater than some bound `eps_sim`. We also assume some concrete probability `guess_pr` for guessing the verifier's challenge in the real-world by sampling it uniformly at random beforehand.

```
axiom single_run_zk (D<:Distinguisher_t{ComChg}) V' vaux paux w x i:
 0 ≤ i < Nsim ⇒
  | Pr [ Distinguish(D).game(i,V',vaux,paux,w,x,false) : res ]
    - Pr [ Distinguish(D).game(i,V',vaux,paux,w,x,true) : res ] | ≤ eps_sim.
```

Here, `Distinguish` is a re-statement of the single-run ZK game to match the operator-based syntax we use for these meta-arguments, and i is (fixed) auxiliary information that is provided to `D` in order to allow generically using it in a hybrid argument (this will encode the hybrid step at which we are using the assumption). The full result we obtain for the ZK property is the following.

```
lemma zk D V' w x: R w x ⇒
| Pr [ ZK(D).game(V',w,x,true) : res ] - Pr [ ZK(D).game(V',w,x,false) : res ] |
    ≤ (1-guess_pr)^Nsim + Nsim*(2*eps_sim).
```

Intuitively, our proof strategy is composing two hybrid arguments: for a single run of the simulator we use a hybrid argument over the party views, and for the meta-theorem we conduct a hybrid argument over the multiple executions of the simulator. However, the fact that each step in the outer hybrid can fail or succeed (and the inner hybrid is only useful in the case of success) complicates the proof significantly. We proceed in a sequence of hops, that first modifies the real world by introducing a bad event that is hidden from the adversary's view and therefore easy to bound: at each execution of the protocol we try to guess the verifier's challenge at random. Once the bad event is in the real game, we conduct a hybrid that gradually changes the real executions to simulated ones. Each step reduces to our single-run assumption, but the analysis of the reduction is complex, as it must address the various cases where the simulation was successful or failed. We believe that our modular proof may be of independent interest.

MPC protocols in next-message syntax We formalize an abstract theory that captures a next-message syntax for MPC protocols, where all parties proceed by synchronous rounds as follows.

```
op local_protocol_round : party → round → public_input →
  local_input → local_rand → in_msgs → pmsgs.

op protocol_round (round:round) (x:public_input) (ws:local_input pmap) (rs:local_rand pmap) (ins:in_msgs pmap
    ) : pmsgs pmap =
  let xs = zip3 ws rs ins in imap (fun i (wi_ri_insi:_*_*_) ⇒
  local_protocol_round i round x wi_ri_insi.`1 wi_ri_insi.`2 wi_ri_insi.`3) xs.

op protocol (x:public_input) (ws:local_input pmap) (rs:local_rand pmap) : trace * local_output pmap = (...)
```

This level of detail is sufficient to capture the notion of pairwise consistent views between a pair of parties and to state/prove Lemma 3.1 given in Section 2.

```
op consistent_views (x:public_input) (i j:party) (vi vj:view) : bool =
  valid_view x vi ∧ valid_view x vj ∧
  valid_rand x (get_view_rand i vi) ∧ valid_rand x (get_view_rand j vj)
  ∧ consistent_inputs x i j (get_view_inputs i vi) (get_view_inputs j vj)
  ∧ get_view_in_msgs j vi = get_view_out_msgs j i x vj
  ∧ get_view_in_msgs i vj = get_view_out_msgs i j x vi.

op valid_inputs x ws =
  ∀ i j, i ∈ partyset ∧ j ∈ partyset ⇒ consistent_inputs x i j ws[i] ws[j]

op consistent_trace x tr : bool =
  ∀ i j, i ∈ partyset ∧ j ∈ partyset ⇒
    consistent_views x i j (get_view i tr) (get_view j tr).

lemma local_global_consistency x tr :
  consistent_trace x tr = (∃ ws rs,
    valid_rands x rs ∧ valid_inputs x ws (protocol x ws rs).`1 = tr).
```

This abstract theory can be applied to any protocol that can be expressed in this syntax. As mentioned in the previous section, we instantiate this result to derive the lemma for our new formalization of Maurer's protocol, as they were developed at the same time. We have not integrated it with the pre-existing BGW formalization, which continues to rely on an axiom for this property. Our library also includes general MPC correctness and security results for abstract circuits, where each gate corresponds to a round in the next-message syntax. Our definitions and proofs for next-message protocols instantiate the abstract theory presented in Section 4.2.

5 Verified Implementations

We divide this section into three parts. First, we briefly describe how we reused the results in [19] to obtain an implementation based on the BGW protocol [11]. We then present a second instantiation where we use Jasmin to obtain an optimized formally verified implementation of Maurer's MPC protocol [27]. We conclude the section with a discussion of the advantages and disadvantages of both approaches wrt to assurance, development time and performance.

The implementations are obtained by using an extended version of the EasyCrypt extraction tool developed in [19]. We have used two approaches to code extraction. The first approach is a complete extraction of the fully instantiated top-level MitH functional operators (modules, imperative procedures and proofs are ignored during extraction). The second approach is an independent extraction of each component (MitH formalization, MPC protocol, commitment scheme, etc.) with pruning to allow for plugging-in optimized implementations of certain operators, when they match verified Jasmin implementations. The main difference between the two is that the fully automatic extraction completely flattens the development, which makes the resulting code hard to manage. The modular extraction allows us to rely on different extraction options for each component, and to easily replace one component with another. The caveat is that the integration of the various components is done by hand. In each instantiation below we explain how the code extraction was conducted.

5.1 Instantiation based on the BGW Protocol

We refine the notion of MPC protocol to the concrete case of secure arithmetic circuit evaluation where parties evaluate addition, multiplication and scalar multiplication gates sequentially.

```
type wire_t = t. type topology_t = int * int * int * int.

type gates_t = [
  | PInput of int | SInput of int | Constant of int & t
  | Addition of int & gates_t & gates_t
  | Multiplication of int & gates_t & gates_t
  | SMultiplication of int & gates_t & gates_t ].

type circuit_t = topology_t * gates_t.
```

The circuit is defined over wires, which are elements of a finite field `t`. The topology (type `topology_t`) is a tuple of integers that fixes the number of public input wires, the number of secret input wires, the number of gates in the circuit, and the number of output wires. Intuitively, when n parties securely evaluate such a circuit, all of them will receive the values of the public input wires in the clear, which is consistent with our assumption in Section 4.2 that all parties receive the same public input. The `gates_t` type permits specifying the different gates that may occur in the circuit (above all gates also carry a gate identifier of type *int*); note that in this formalization of the BGW protocol a circuit for a boolean function is specified as a value in an inductive type, which is essentially a tree: the output gate is the root, nodes correspond to arithmetic gates, and the input gates form the leaves.

We define the secure evaluation of a circuit by fixing a secret sharing scheme and a set of low-level protocols that can be used to compute the above arithmetic gates over secret shares.[6] The `sinput_t` type for secret inputs is defined as a list of finite field values, and the secret input gate performs a fresh secret sharing. The BGW protocol is obtained by instantiating the secret sharing scheme with Shamir's secret sharing and the low-level arithmetic gate protocols (including the randomization output gate *refresh*) proved secure in [19].

The output of the protocol is computed by performing a share-rerandomization step followed by an explicit unsharing where all parties publish the final shares of the output wires. This allows for a compositional proof, where simulators for low-level gates can be combined modularly to obtain a simulator for the entire protocol. This compositional property is studied in [14, 3] and its formalization was adapted from [19].

We briefly describe how correctness and security of the full protocol are proved in EasyCrypt. Correctness is proved by induction on the structure of the circuit and relying on the correctness of the low-level gates at each inductive step. The 2-privacy of the protocol is obtained by instantiating the secure composition theorem, assuming that the low-level arithmetic gates guarantee a weak notion of security we call t-pre-output-privacy and that the refresh gate satisfies t-privacy. We note that the tree/inductive-type representation of circuits we use in the BGW instantiation allowed for simpler proofs, as it allows directly applying the EasyCrypt logic to perform inductive reasoning over the structure of the circuit. However, we concluded that this introduces an unnecessary abstraction gap to more efficient implementations that see circuits as a list of gates (under some topological sorting). Our second instantiation, which we describe in the next subsection, uses the latter approach but still retains the modularity on arithmetic gates.

For this instantiation based on BGW we initially performed a fully automatic extraction using Pedersen commitments, which exactly matches the implementation in [19]. Field operations and basic data structures were mapped to unverified OCaml libraries. As a first optimization, we formalized a PRF-based commitment scheme following [21], which upon extraction we map to an unverified HMAC implementation. This is the implementation for which we collect performance data at the end of this section.

[6] Recall that in the MitH construction these secret input values will be an additive splitting of a witness, but this should not be confused with the secret sharing performed inside the MPC protocol that we consider in this section.

5.2 Instantiation based on Maurer's Protocol

We developed a new Jasmin implementation of the arithmetic gates used by the passively secure variant of Maurer's protocol [27]. Again, we consider the specific case of 5 computing parties, which can be plugged into the abstract MitH construction we discussed earlier as an alternative to the BGW instantiation.

Simultaneously, the full MPC protocol is also specified in EasyCrypt, following the next-message syntax we introduced in Section 4.4 and a list-based representation of the arithmetic circuit. The proofs of correctness and security of the protocol in EasyCrypt use the same overall strategy as that described in the previous section (with induction performed over the list structure of the circuit, which requires slightly more involved invariants that make explicit the state kept by parties). However, we introduced two simplifications in the resulting ZK protocol to illustrate how optimizations at the cryptographic design level can be carried out with reasonable effort. The remaining differences in the proofs are due to syntactic definitional choices made to allow for an easy and efficient integration of the Jasmin gates. We first give a short overview of the improvements obtained at the cryptographic design level, and then we explain more in detail how we connect formal verification results obtained for the Jasmin implementation with the code extraction mechanism in EasyCrypt.

Simplification of the MitH transformation

We first modify the secret sharing step performed by the prover prior to running the MPC protocol, so that it uses the same secret sharing scheme used by the MPC protocol. This removes the need to use input gates and perform an oblivious unsharing of the additive splitting within the MPC protocol; however, it requires a slight modification to the proof of soundness of the MitH construction to obtain the same result: one needs to take advantage of the fact that the verifier checks for well-formedness of inputs to the MPC protocol to guarantee that one catches the prover when providing an invalid secret sharing. The second simplification removes the final step of resharing the MPC protocol outputs and directly proves that this is not necessary for the specific setting in which the MPC protocol is run within the MitH construction. This is because the simulator has enough information in the specific case of 2-out-of-5 corrupted parties to complete the simulated views without the need to introduce extra randomness (ZKBoo [20] performs a similar optimization). These simplifications rely on well-known properties of Maurer's protocol and do not have a huge impact in performance, but we believe that they further support our claim that our approach extends to other protocols of comparable design and proof complexity.

A stateful arithmetic gate API

Our EasyCrypt specification of Maurer's protocol relies on the following gate-level operators:

```
op mul_start(wi wj : wid, wst : wire_st, r : rand) : msgs =
  let (w,wires) = wst in let swi = oget wires[wi] in let swj = oget wires[wj]
  in share (cross swi swj) r.

op mul_end(ms : msgs, wst : wire_st) : wire_st =
  let (w,wires) = wst in
  let ss = add_shr ms[0] ms[1] in let ss = add_shr ss ms[2] in
  let ss = add_shr ss ms[3] in let ss = add_shr ss ms[4] in
           (w + 1, wires[w ← ss]).

op add(wi wj : wid, wst : wire_st) : wire_st =
  let (w,wires) = wst in
  let ash = add_shr (oget wires[wi]) (oget wires[wj]) in (w+1,wires[w ← ash]).
```

These operators permit an orchestrator, e.g., the prover in the MitH construction, to execute the whole protocol in rounds. For an addition gate, only a call per party is needed, since the `add` gate does not involve communications (this is the same for the scalar multiplication gate which we omit for brevity). Multiplication gates execute in three steps: the `mul_start` operator must first be called for all parties, at which point they all define the messages to be sent in that round; this is then followed by a call to a `dispatch` routine that rearranges the transmitted messages so that they are accessible to the intended parties; finally, the `mul_end` operator must be called for all parties to conclude the operation and consolidate the local states. Indeed, these operators model stateful parties evaluating a circuit gate-by-gate (i.e., wire-by-wire): on each step they take a current state which includes secret shares for all previously computed wires and possibly some randomness. When the round is completed they add a new secret share for the wire that was just computed.

The sharing of inputs, which is carried out by the prover is modelled using an input gate, whereas the reconstruction of the output by all parties is modelled using an output gate.

```
op input_start(v : val, r : rand) : msgs = share v r.
op input_end(m : msg, wst : wire_st) : wire_st =
    let (w,wires) = wst in (w + 1, wires[w ← m]).

op output_start(wo : wid, wst : wire_st) : msgs =
    let (w,wires) = wst in Array5.create (oget wires[wo]).

op output_end(ms : msgs) : val = unshare ms.
```

Here, the `share` operator takes a finite field element (the secret) and randomness corresponding to 9 more finite field elements. It performs an additive secret sharing that splits the secret into 10 parts and then constructs secret shares for all parties by providing them with a subset of 6 of those parts. This assignment guarantees that any set of 3 or more parties can reconstruct the secret, but 2 shares reveal nothing about the secret. Moreover we carefully tailored the way in which shares are stored by each party, so that a single implementation of the code for gates could work for all parties, independently of the party number. This greatly simplified the implementation effort and the verification effort as we describe next. In the output gate, parties simply send their shares to each other and perform the unsharing by recovering the 10 parts of the secret and adding them. The specification and proofs for gate correctness and security are fully generic wrt to the prime that defines the finite field.

A verified Jasmin implementation The Jasmin implementation offers an interface which matches the gate operators shown above. We give here the Jasmin entry points

```
fn add5(reg u64 status w1 w2 curwire) → reg u64
fn mult_start5(reg u64 status w1 w2 outI randomnessI)
fn mult_end5(reg u64 all_messages status curwire) → reg u64
```

and the corresponding C declarations

```
uint64_t add5(uint64_t*,uint64_t,uint64_t,uint64_t);
void mult_start5(uint64_t*,uint64_t,uint64_t,uint64_t*,uint64_t*);
uint64_t mult_end5(uint64_t*,uint64_t*,uint64_t);
```

which allow high-level code to call the Jasmin-generated assembly.

The state is passed in as a pointer, whereas the input wire numbers are simply integers stored in registers. We have a proof of functional correctness that gates implemented in Jasmin are correct with respect to the EasyCrypt operator that is used in the high-level formalization. These proofs are created in EasyCrypt over a representation of the Jasmin program semantics. We give here

an example correctness lemma, where the hypotheses establish well-formedness conditions on the calling arguments and the initial state of the memory `mem`, as per the inlined comments.

```
lemma add5_correct_pr mem st (cwire wr1 wr2 : int) (wst : wire_st) :
   wst.1 = cwire ⇒ (* cwire = correct number of wires *)
   elems (fdom wst.2) = iota_ 0 cwire ⇒ (* the state is well formed *)
   0 ≤ wr1 < cwire ⇒ 0 ≤ wr2 < cwire ⇒ (* valid input wires *)
   good_wire st cwire ⇒ (* valid memory region to write to *)
   good_wire_shares mem st cwire ⇒ (* valid memory region to read from *)
   wst = lift_state_mem mem st cwire ⇒ (* region stores state *)
 hoare [
  M.add5 : Glob.mem = mem ∧ to_uint status = st ∧ to_uint curwire = cwire ∧
  to_uint w1 = wr1 ∧ to_uint w2 = wr2 ⇒
  good_wire_shares Glob.mem st (cwire + 1) ∧
  lift_state_mem Glob.mem st (cwire+1) = add wr1 wr2 wst ∧
  touches mem Glob.mem (st+cwire*6*8*L) 6 ].
```

The `hoare` claim establishes a Hoare triple that relates the Jasmin implementation `M.add5` to the functional specification `add`. This triple states that if the program starts from an initial memory that encodes some initial party internal state `wst` and input wires `wr1` and `wr2`, then the final memory will encode `add wr1 wr2 wst`, i.e., the output state will be computed according to the specification of the addition gate. Furthermore, the statement guarantees that the memory remains unchanged, except for the updating of the party's state (this is the `touches` predicate, where `L` denotes to the size of a finite field element in memory, `st` is a pointer to the state of the party in memory, and `cwire` is the number of wire shares already stored in the state). Our Jasmin implementations are also modular with respect to the underlying field operations. We have reused a verified implementation implementation of the field $\mathbb{F}_{2^{255}-19}$ and created verified implementations of field operations for small moduli (fitting into a 64-bit word), including a specialized variant for boolean circuits (this uses bitwise operations for computations over $\mathbb{F}_2$).

Extraction and integration We followed a semi-automatic extraction approach and pruned the resulting OCaml code at the operators and data types that match the interface of the Jasmin code. This means that the unverified part of the code is reduced to basic OCaml data types, and to C wrappers between OCaml code and Jasmin code that we have written by hand. This code takes care of memory allocation, randomness generation and conversion between OCaml representations and the input/output memory regions used by the Jasmin routines. Concretely, memory regions used by Jasmin are seen by the OCaml code as a single address (when their size is fixed, e.g., for a message) or as an address/size pair (for variable length structures such as the states of parties). For efficiency purposes, we allocate memory upfront, by instrumenting the C routines that sample the randomness for the protocol.

Finally, we instantiated the commitment scheme with a verified implementation of SHA-3 taken from [4], so overall this implementation removes all cryptography-specific code from the trusted computing base. Computing the commitment over a view can be done simply by passing the memory region that contains the view to this verified routine. C routines for comparing memory regions are provided to allow the verifier checks for commitment correctness and message consistency.

5.3 Discussion

To better estimate the size of our development, Figure 1 provides numbers of lines of code for the EasyCrypt proofs. Among the two instantiations, roughly 60% of the proof effort is shared via abstract components (these numbers do not take into account common concrete components such as hash-based commitment schemes). About 50% of the proof effort for the Maurer instantiation concerns the functional correctness of the Jasmin gates.

Small Modulus							
Maurer				BGW			
#gates	100	1000	10000	#gates	100	1000	10000
$\mathcal{P}$ MPC	1,14	88,78	8728,81	$\mathcal{P}$ MPC	4,15	123,32	10540,82
$\mathcal{P}$ Com	0,87	4,28	39,47	$\mathcal{P}$ Com	0,57	6,30	132,50
$\mathcal{P}$ Total	2,15	93,87	8776,08	$\mathcal{P}$ Total	5,35	135,90	10759,47
$\mathcal{V}$ CVeriv	0,20	1,53	13,83	$\mathcal{V}$ CVerif	0,23	2,46	49,92
$\mathcal{V}$ Check	0,46	35,81	3491,19	$\mathcal{V}$ Check	3,25	196,14	25794,72
$\mathcal{V}$ Total	0,66	37,34	3505,02	$\mathcal{V}$ Total	3,48	198,60	25844,64

Large Modulus							
Maurer				BGW			
#gates	100	1000	10000	#gates	100	1000	10000
$\mathcal{P}$ MPC	1,35	92,96	9031,57	$\mathcal{P}$ MPC	7,05	150,67	11053,62
$\mathcal{P}$ Com	2,16	15,04	141,75	$\mathcal{P}$ Com	1,34	18,22	363,61
$\mathcal{P}$ Total	3,83	111,09	9195,54	$\mathcal{P}$ Total	9,09	176,03	11498,90
$\mathcal{V}$ CVeriv	0,69	5,88	55,06	$\mathcal{V}$ CVerif	0,51	7,12	134,33
$\mathcal{V}$ Check	0,65	49,56	4819,85	$\mathcal{V}$ Check	5,60	232,10	25757,51
$\mathcal{V}$ Total	1,33	55,44	4874,91	$\mathcal{V}$ Total	6,11	239,22	25891,84

Table 1: Benchmarking results for random circuits of indicated size. Small modulus variants relied on a prime $q = 2^{61} - 1$ that fits in to a 64-bit word, whereas the large modulus results uses $q = 2^{255} - 19$. All times are given in milliseconds. For the prover we report times for computing the MPC protocol in the head and committing to the views. For the verifier we report times for verifying the commitments and checking consistency between views. Data was collected using a modest 2.3 GHz Quad-Core Intel Core i7 with 32 GB RAM, 512 KB L2 CACHE PER CORE, 8 MB L3 CACHE

The two implementations we describe in this section explore two different approaches for obtaining verified implementations from complex EasyCrypt specifications. The first one uses an automatic extraction of the whole development to OCaml, which means that the target code is a simple syntactic translation of the functional EasyCrypt operators. The main advantage of this approach is the reduced development time; the main disadvantages are code management and performance. We also rely on unverified OCaml libraries for finite-field and low-level cryptography, which increases the TCB. In our second approach, we have a meet in the middle strategy: we implement the low-level components in Jasmin, which reduces the TCB but significantly increases the development time (this can of course be amortized, as we did, by reusing existing verified code for the finite-field $\mathbb{F}_{2^{255}-19}$ and hash function). We use a semi-automated extraction strategy that preserves the modular structure of the EasyCrypt proofs, but makes the extracted code roughly 3x larger than the BGW one due to structural redundancies. The code resulting from extraction is now conceptually much simpler, as we only generate the code that orchestrates calls to the MPC protocol gates, commitment scheme, etc. However, we need to write custom binding code in C to connect the Jasmin code to the OCaml code (about 15% of the executable code). Roughly another 15% of the executable Maurer code concerns Ocaml bindings for the Jasmin code and for unverified Ocaml libraries for basic data structures. In the future, this code could be automatically generated from meta-information that exists in the EasyCrypt formalization, and this is an interesting direction for future work.

Table 1 gives a comparison of the performance of the two implementations for a growing number of gates; the input circuits are generated at random with essentially the same number of multiplication and addition gates. We note that in MitH, the difference between multiplication and addition

gates is not as dramatic as in standard MPC, since there is no latency associated with communications. For example, the difference between addition and multiplication in our implementations is 5%, since the computation time seems to be dominated by memory access and the overhead of interfacing extracted code with low-level implementations.

The performance benefits of introducing the optimized Jasmin implementations are clear: as the number of gates increases, the performance improvement increases as well, leading to an 18% improvement in prover time and a 5x improvement in verifier time for the larger circuits The heavier computations take place on the verifier side, when verifying consistency between views. The fact that we keep the orchestration of the consistency checks in extracted code means that we do not get the opportunity to hand-optimize this step in either protocol. However, the implementation choices in the BGW version (intensively using list-based operations) clearly lead to an extracted code that performs quite poorly.

The proof size for our implementations includes the initial commitments ($160 = 5 * 32$ bytes), plus the two opened views. We have not implemented any compression techniques, so our implementation is naive in this respect. In the Maurer implementation, a sharing takes $L*8*5*6$ bytes, where L is the size of the representation of the prime q and 8 is the size of the processor word. This means that the view includes $L*8*(9+5*6)$ bytes per multiplication gate (here 9 is the number of random field elements required to perform a secret sharing and 5 is the number of parties), plus $L*8*6$ bytes per secret input.

The timings we report seem poor in comparison to those obtainable with a highly optimized implementation such as ZKBoo [20] where, in addition to improvements in the protocol design (e.g., adopting a 2-out-of-3 trust model rather than a 2-out-of-5) there is a lot of work in fine-tuning the implementation and use of parallelism. There, for example, carrying out a proof for a circuit with roughly 30K gates (e.g., SHA-1) is reported as taking 13ms in prover time and 5ms for verifier time (and this including repetition for reducing the soundness error, which can be parallelized for non-interactive proofs), i.e. roughly three orders of magnitude faster than the times we report.

However, we do not think that this is indicative that our approach intrinsically leads to prohibitive execution times. For example, in the same paper, the Pinocchio [28] prototype is used as baseline; it is reported that the improvement in prover time that ZKBoo offers is precisely 3 orders of magnitude, which would place Pinocchio much closer to the performance of our implementations. Pinocchio implements a different family of ZK protocols, but it shares many similarities with our prototype; in particular, its goal is to demonstrate the potential of a new technology, rather than exploring its performance limits. Moreover, the Pinocchio implementation is fully generic, and it can compute any circuit, rather than being specifically fine-tuned for a particular family of algorithms or data structure.

We also briefly mention the results presented in [16], which extends MitH to use MPC protocols with preprocessing. There, a prover time of 851ms is reported for computing a ZK proof involving 10K gates (also already accounting for soundness error reduction). Again, this is significantly better than the values we report, but it does show that variability in performance results for such frameworks is quite large, depending on many factors.

We believe that many optimizations can be applied to improve the execution time of our formally verified implementations of MitH very significantly, even though the use of a functional language as the target for extraction will always introduce some overhead. Indeed, an important direction for future work is to see how close to the performance of aggressively optimized unverified implementations one can get.

6 Related Work

ZK protocols are a fast-moving area within cryptography, and many protocols exist both for specific proof goals and for settings that require a flexible solution that can be used for any relation. Only a very small part of this field has been studied from the perspective of computer-aided cryptography [7]. The work in [8] was the first to formalize a special class of Σ^{ϕ}-protocols in CertiCrypt, a predecessor of EasyCrypt implemented as a Coq library, and to prove the security of general *and* and *or* composability theorems for Σ^{ϕ}-protocols. The more recent work from [15] restates many of these results for Σ-protocols in CryptHOL. It additionally formalizes abstract and concrete commitment scheme primitives and proves a construction of commitment schemes from Σ-protocols.

The most significant machine checked endeavor for ZK is the work in [6], that developed a full-stack verified framework for developing ZK proofs. The framework encompasses a non-verified optimizing ZK compiler that translates high-level ZK proof goals to C or Java implementations, and a verified compiler that generates a reference implementation. The machine checked effort lies in proving that, for any goal, the reference implementation satisfies the ZK properties and that the optimized implementation has the same observable behavior as the reference implementation. The core of the verified compiler builds on top of the results from [8], extended with *and* compositions of $\Sigma^{\mathtt{GSP}}$-protocols, and generates CertiCrypt proof scripts for automatically proving the equivalence of the two implementations.

There is nowadays a vast body of MPC protocols and frameworks, some of which have been formally verified using machine-checked tools. CircGen [1] is a verified compiler that translates C programs into boolean circuits by extending the CompCert C compiler with an additional backend translation to Boolean circuits. This back-end can then be used to feed to an EasyCrypt machine-checked implementation of Yao's 2-party secure function evaluation protocol. The work in [24] formalizes in EasyCrypt the n-party MPC protocol due to Maurer [27] for the actively secure case. In this paper we formalize in EasyCrypt the passive case, and also provide a formally verified Jasmin implementation. The work in [19] provides verified implementations of proactively secure MPC, including an EasyCrypt formalization of the BGW [12] MPC protocol for passive and static active adversaries that we adapt and build on in this paper.

Independent work on verifying MPC-in-the-HEAD. Recently, in independent work [30], Sidorenco, Oechsner and Spitters presented a machine-checked security proof for a class of Σ-protocols that follows the approach to MitH introduced by the ZKBoo protocol [20], which is an important optimized derivative of the MitH paradigm. The authors give a formalization of *decomposition protocols* and show how they can be modularly used to construct Σ-protocols, which are secure in the sense of special-soundness and special honest verifier zero-knowledge. We note that these properties are specific to Σ protocols; indeed, additional transformations and security proofs are needed to obtain the standard non-interactive proof-of-knowledge guarantees that these protocols provide.

The contributions in this paper compare to those in [30] as follows. We also consider 3-pass ZK protocols, but we give *both* a machine-checked proof of security for MitH *and* a formally verified implementation; our implementation of MitH includes verified implementations for the underlying MPC, secret sharing and commitment sub-protocols and can be used in practice to prove arbitrary goals in zero knowledge. Second, our formalization follows the original IKOS construction given in [25] and uses the standard syntax and security notions for zero-knowledge proofs, MPC protocols and commitment schemes. This has the advantage of allowing us to build on and to deploy standard components, but introduces the challenge of formalizing more complex security proofs. For example,

the proof of the (malicious verifier) zero knowledge property is quite challenging when compared to the honest verifier variant because the distribution of the verifier's challenge is not known a priori and a form of rejection sampling must be used in the simulation. The techniques we used to establish this result allowed us also to formalize the reduction of the soundness error by repetition and we are currently working to extend this result for the zero-knowledge property of the sequential composition construction.

References

[1] 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, 2017.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 965–982. IEEE, 2020.

[3] 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.

[4] José Bacelar Almeida, Cécile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. Machine-checked proofs for cryptographic standards: Indifferentiability of sponge and secure high-assurance implementations of SHA-3. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1607–1622. ACM, 2019.

[5] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. 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 2087–2104. ACM, 2017.

[6] José Bacelar Almeida, Manuel Barbosa, Endre Bangerter, Gilles Barthe, Stephan Krenn, and Santiago Zanella Béguelin. Full proof cryptography: verifiable compilation of efficient zero-knowledge protocols. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 488–500, 2012.

[7] Manuel Barbosa, Gilles Barthe, Karthikeyan Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. *IEEE Security and Privacy*, 2021.

[8] Gilles Barthe, Daniel Hedin, Santiago Zanella Béguelin, Benjamin Grégoire, and Sylvain Heraud. A machine-checked formalization of sigma-protocols. In *2010 23rd IEEE Computer Security Foundations Symposium*, pages 246–260. IEEE, 2010.

[9] Carsten Baum, Cyprien Delpech de Saint Guilhem, Daniel Kales, Emmanuela Orsini, Peter Scholl, and Greg Zaverucha. Banquet: Short and fast signatures from AES. In Juan A. Garay, editor, *Public-Key Cryptography - PKC 2021 - 24th IACR International Conference on Practice and Theory of Public Key Cryptography, Virtual Event, May 10-13, 2021, Proceedings, Part I*, volume 12710 of *Lecture Notes in Computer Science*, pages 266–297. Springer, 2021.

[10] Carsten Baum and Ariel Nof. Concretely-efficient zero-knowledge arguments for arithmetic circuits and their application to lattice-based cryptography. In *IACR International Conference on Public-Key Cryptography*, pages 495–526. Springer, 2020.

[11] 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.

[12] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 351–371. 2019.

[13] Rishabh Bhadauria, Zhiyong Fang, Carmit Hazay, Muthuramakrishnan Venkitasubramaniam, Tiancheng Xie, and Yupeng Zhang. Ligero++: A new optimized sublinear iop. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 2025–2038. Association for Computing Machinery, 2020.

[14] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *Proceedings of the 13th European Symposium on Research in Computer Security*, pages 192–206. Springer, 2008.

[15] David Butler, Andreas Lochbihler, David Aspinall, and Adrià Gascón. Formalising ς-protocols and commitment schemes using CryptHOL. *Journal of Automated Reasoning*, pages 1–47, 2020.

[16] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1825–1842. ACM, 2017.

[17] Cyprien Delpech de Saint Guilhem, Lauren De Meyer, Emmanuela Orsini, and Nigel P. Smart. BBQ: using AES in picnic signatures. In Kenneth G. Paterson and Douglas Stebila, editors, *Selected Areas in Cryptography - SAC 2019 - 26th International Conference, Waterloo, ON, Canada, August 12-16, 2019, Revised Selected Papers*, volume 11959 of *Lecture Notes in Computer Science*, pages 669–692. Springer, 2019.

[18] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, and Titouan Tanguy. Limbo: Efficient zero-knowledge mpcith-based arguments. Cryptology ePrint Archive, Report 2021/215, 2021. https://eprint.iacr.org/2021/215.

[19] Karim Eldefrawy and Vitor Pereira. A high-assurance evaluator for machine-checked secure multiparty computation. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and*

Communications Security, CCS 2019, London, UK, November 11-15, 2019, pages 851–868. ACM, 2019.

[20] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. Zkboo: Faster zero-knowledge for boolean circuits. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 1069–1083. USENIX Association, 2016.

[21] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*, volume 10403 of *Lecture Notes in Computer Science*, pages 66–97. Springer, 2017.

[22] Yaron Gvili, Julie Ha, Sarah Scheffler, Mayank Varia, Ziling Yang, and Xinyuan Zhang. Turboikos: Improved non-interactive zero knowledge and post-quantum signatures. In Kazue Sako and Nils Ole Tippenhauer, editors, *Applied Cryptography and Network Security - 19th International Conference, ACNS 2021, Kamakura, Japan, June 21-24, 2021, Proceedings, Part II*, volume 12727 of *Lecture Notes in Computer Science*, pages 365–395. Springer, 2021.

[23] Helene Haagh, Aleksandr Karbyshev, Sabine Oechsner, Bas Spitters, and Pierre-Yves Strub. Computer-aided proofs for multiparty computation with active security. In *Proceedings of the 31st Computer Security Foundations Symposium*, page In print. IEEE.

[24] Helene Haagh, Aleksandr Karbyshev, Sabine Oechsner, Bas Spitters, and Pierre-Yves Strub. Computer-aided proofs for multiparty computation with active security. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 119–131. IEEE, 2018.

[25] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, pages 21–30. ACM, 2007.

[26] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 525–537. ACM, 2018.

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

[28] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 238–252, 2013.

[29] Okan Seker, Sebastian Berndt, Luca Wilke, and Thomas Eisenbarth. Sni-in-the-head: Protecting mpc-in-the-head protocols against side-channel analysis. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1033–1049. ACM, 2020.

[30] Nikolaj Sidorenco, Sabine Oechsner, and Bas Spitters. Formal security analysis of mpc-in-the-head zero-knowledge protocols. Cryptology ePrint Archive, Report 2021/437, 2021. https://eprint.iacr.org/2021/437.