Boosting the Performance of High-Assurance Cryptography: Parallel Execution and Optimizing Memory Access in Formally-Verified Line-Point Zero-Knowledge

Samuel Dittmer[1], Karim Eldefrawy[2], Stéphane Graham-Lengrand[2], Steve Lu[1], Rafail Ostrovsky[3], and Vitor Pereira[2]

[1]Stealth Software Technologies, Inc., Los Angeles, CA, USA
[2]SRI International, Menlo Park, CA, USA
[3]University of California, Los Angeles, CA, USA

May 13, 2024

Abstract

Despite the notable advances in the development of high-assurance, verified implementations of cryptographic protocols, such implementations typically face significant performance overheads, particularly due to the penalties induced by formal verification and automated extraction of executable code. In this paper, we address some core performance challenges facing computer-aided cryptography by presenting a formal treatment for accelerating such verified implementations based on multiple generic optimizations covering parallelism and memory access. We illustrate our techniques for addressing such performance bottlenecks using the Line-Point Zero-Knowledge (LPZK) protocol as a case study. Our starting point is a new verified implementation of LPZK that we formalize and synthesize using EasyCrypt; our first implementation is developed to reduce the proof effort and without considering the performance of the extracted executable code. We then show how such (automatically) extracted code can be optimized in three different ways to obtain a **3000x speedup** and thus matching the performance of the manual implementation of LPZK of [18]. We obtain such performance gains by first modifying the algorithmic specifications, then by adopting a provably secure parallel execution model, and finally by optimizing the memory access structures. All optimizations are first formally verified inside EasyCrypt, and then executable code is automatically synthesized from each step of the formalization. For each optimization, we analyze performance gains resulting from it and also address challenges facing the computer-aided security proofs thereof, and challenges facing automated synthesis of executable code with such an optimization.

Keywords: Zero-Knowledge; Formal Verification; Parallelism; Verified Implementation; Verified Optimizations; Code Synthesis

A shorter version of this paper will appear in the Proceedings of the 2023 ACM Conference on Computer and Communications Security (CCS 2023). This is the full version.

Contents

1 Introduction

Developing high-assurance, verified implementations of cryptographic primitives and protocols has attracted a lot of recent attention [8] because it can minimize implementation and security bugs in critical cryptographic code. This is achieved through the development and application of formal, machine-checkable approaches to the design, analysis, and implementation of cryptographic primitives and protocols. The broader goal of computer-aided cryptography [8] is not only to provide a better way to tame the complexity of security proofs, but also to make sure that the proven security properties are preserved in concrete implementations of cryptographic algorithms.

Unfortunately, such high-assurance implementations have so far suffered from significant performance overheads, particularly those that follow an approach based on formal verification and automated extraction of executable code. We observe that there are some core performance challenges facing computer-aided cryptography, and our goal is to address these performance challenges by presenting a formal treatment on how such verified implementations can be accelerated using multiple optimizations covering parallelism and memory access. We illustrate our techniques for addressing these performance bottlenecks using a concrete zero-knowledge (ZK) protocol as a case study. ZK protocols allow a prover $\mathcal{P}$, with input (x, w), and a verifier $\mathcal{V}$, with input x, to jointly compute a boolean function $f(x, w)$ that accepts the proof if $R(x, w)$ holds, for an NP relation R. First introduced in seminal work by Goldwasser, Micali, and Rackoff [23], ZK protocols and subsequent applications have seen significant improvements over the past decades, with the proposal of several efficient, scalable, and practical constructions for generic relations or the use of ZK protocols in different privacy-preserving application scenarios.

Despite being a fast-growing research area, only a small subset of ZK protocols has been studied from the perspective of computer-aided cryptography. Works on the connection between computer-aided cryptography and ZK include [1, 28, 6], where the authors developed comprehensive computer-aided security proofs and also high-assurance mechanisms for the compilation and evaluation of ZK statements. Concretely, the focus of [28] and [6] was the MPC-in-the-Head (or “IKOS”) paradigm [25], a modular construction that yields ZK protocols from secure multiparty computation (MPC) in combination with commitment schemes. Nevertheless, the performance obtained by the verified implementation of IKOS, whilst not being prohibitive, is still far from the efficiency obtained by optimized unverified implementations. In fact, perhaps the biggest critique of computer-aided cryptography, namely the approaches that focus on the production of high-level verified implementations via code extraction, is that executable code derived from it is not as efficient as unverified custom implementations. Note that this statement is not entirely true for other computer-aided cryptography approaches that focus on low-level verified implementations (such as the ones based on Jasmin [2], HACL$\star$ [34] or Vale [15]), who are able to achieve performance on-par with unverified implementations. However, these approaches entail significantly more complexity and tool expertise than our work.

Another efficient class of ZK protocols is derived from vector oblivious linear evaluation (VOLE) constructions [16]. Briefly, a VOLE protocol can be used to establish secret correlated randomness between the prover and the verifier that can then be consumed by the ZK protocol. By using such pre-processing, it is possible to construct fast ZK protocols based on arithmetic field operations. Example of such protocols include Line-Point Zero Knowledge (LPZK) protocol [19, 18], Wolverine [29], Quicksilver [31], and Mac'n'cheese [12].

In LPZK, the prover produces a proof by encoding the witness as an affine line $\mathbf{v}(w) = \mathbf{a}w + \mathbf{b}$, and the verifier checks validity of the proof by querying the line at a single point α. The attractiveness of LPZK lies in its efficiency. Namely, proving satisfiability of (generic) arithmetic circuits requires only 2-5 times the computation of evaluating the circuit in the clear and the prover only communicates roughly 2 field elements per multiplication gate, making LPZK also attractive from a network point of view. LPZK is defined over the *designated verifier non-interactive zero-knowledge* (DVNIZK) model, where it is assumed that input-independent correlated randomness has been pre-processed and that the interaction between the prover and verifier consists of a single message sent by the former to the latter. The terminology *designated verifier* is used to attest that only a verifier holding correlated randomness with the prover is able to verify the proof produced by the prover.

In this work, we develop an end-to-end machine-checked implementation of LPZK, that was obtained from a series of (also machine-checked) optimization steps. In more detail, we start with a *reference* implementation of LPZK, for which we perform the machine-checked completeness, soundness, and zero-knowledge proofs. Next, we apply the following optimization steps:

1. Optimization based on algorithmic re-design: where we optimize the verifier specification
2. Optimization based on the execution model: where we explore the usage of parallelism to speed up the computation
3. Optimization based on memory management: where we replace and augment the data structures used to store input and randomness values in order to reduce access overhead

For each optimization, we prove that the resulting new implementation has the same *observable behavior* as the reference implementation, thus achieving the same level of security. Our goal is to start with an LPZK specification that minimizes the proof effort required to machine-check the completeness, soundness and zero-knowledge properties, and then introduce optimizations (potentially increasing the complexity of the protocol specification) without the need to re-implement the security proof in each optimization step.

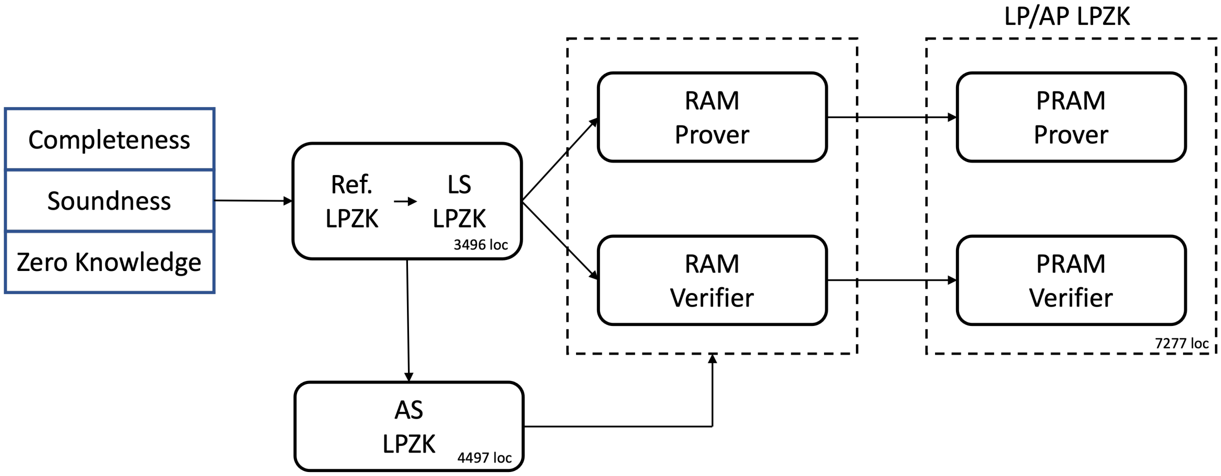

Figure 1: Overview of our formalization, annotated with overall lines of proof code (denoted by *loc*). RAM stands for Random Access Memory machine and PRAM stands for Parallel RAM. LPZK denotes Line-Point Zero-Knowledge. *LS LPZK* denotes a list-based sequential LPZK, whilst *AS LPZK* denotes an array-based one. *LP LPZK* denotes a list-based LPZK in the PRAM model while *AP LPZK* denotes an array-based LPZK in the PRAM model.

Contributions Our contributions are summarized in Figure 1. Our starting point is a new *reference* EasyCrypt formalization of the LPZK protocol. As a first optimization step, we modify the verifier algorithm to reduce the number of circuit iterations performed, achieving new LPZK implementation for which we obtain executable code, that we denote by *LS LPZK*, denoting a list-based sequential implementation. This implementation is then optimized by utilizing the parallel capabilities of LPZK and by introducing different memory access data types. To achieve a parallelized LPZK implementation, we first define Random Access Memory (RAM) machine wrappers both for the prover and for the verifier. These RAM wrappers are then used to formalize a parallelized LPZK implementation under the parallel RAM (PRAM) model proposed in [21], that we dub *LP LPZK*, standing for list-based parallel implementation. Finally, we modified the memory structures of the list-based implementation to achieve an array-based sequential LPZK implementation, that we portray as *AS LPZK*, which can also be matched against our parallelism methodology (*AP LPZK*). The final optimized version exhibits up to **3000x speedup** and matches the performance of the manually implemented version of LPZK give in [18].

Our parallel implementation was derived via a new parallelism EasyCrypt framework, that is of independent interest. This framework was developed based on three main ideas: i. support any (parallelizable) algorithm and data structures; ii. support any number of parallel cores; and iii. achieve a high degree of modularity, in the sense that the parallel model formalization was done *once and for all* and new secure parallel executions of cryptographic algorithms could be obtained with reduced proof effort, without the need to repeat the proof every time the framework is instantiated with a new cryptographic protocol. Briefly, our parallelism library requires the user to provide a methodology to split the *circuit* being evaluated and to aggregate the resulting smaller *circuits* back to the original one. Our library then yields secure parallel evaluations of the desired cryptographic primitive according to the split-and-aggregate model created by the split and aggregate functions, with the user being required to prove only some side-conditions that these two functions need to follow.

In summary, at a high-level, we developed the first verified, high-assurance implementation of LPZK with performance matching an unverified implementation of the same protocol. Our formal work was verified in EasyCrypt and leverages a previously developed IKOS-based ZK formalization [6]. The verified optimized executable code is obtained via the OCaml extraction approach first used to extract verified MPC executable code [20], and later an executable for a ZK protocol based on MPC-in-the-head [6]. We also make contributions to the general field of computer-aided cryptography and to the EasyCrypt tool-set, as follows:

- We demonstrate the feasibility of optimizing verified code, by introducing new techniques to reason about observable behavior of implementations. We show that a significant portion of the proof effort is concentrated around the *reference* specification and that deriving the security results of the optimized versions can be obtained with reduced proof overhead

- We develop a modular framework for parallelism in EasyCrypt, that is capable of generating parallel descriptions of EasyCrypt specifications for an arbitrary number of cores. Our parallelism formalization tolerates any (parallelizable) *circuit* and only requires a user to provide a function that splits the *circuit* into smaller, independent, circuits, and a function that correctly aggregates the outputs of the cores

- We improve the EasyCrypt extraction mechanism first defined in [20] and refined in [6] to capture the parallel execution model, as well as the memory-based optimizations that are employed in this work

Technical Challenges The challenges facing our work can be summarized into three categories: machine-checked security proofs, high-assurance cryptographic implementations, and the secure optimization of (executable) cryptographic code.

Machine-checked security proofs. Despite notable advances in the development of more comprehensive tools and proof methodologies, machine-checked cryptography still faces many challenges, particularly regarding the required tool proficiency in order to perform such machine-checked proofs. Concretely to the ZK context, these challenges are further exacerbated if one takes into account the formalization of foundational results such as soundness or zero-knowledge. In addition, machine-checking a security proof is a process that needs to take into account all the nuances that are sometimes skipped in pen-and-paper proofs, making them more expensive and time-consuming in terms of the required human effort.

High-assurance cryptographic implementations: Once a formalization is complete, another challenge resides on how to obtain software that securely implements the specification of the cryptographic primitive or protocol that was machine-checked. In other words, how to develop an implementation that mirrors the formalization and for which there are guarantees that the functional correctness and security properties proven also apply to the actual executable code. We address this challenge by relying on the EasyCrypt code extraction pipeline proposed in [20] and refined in [6]. This approach uses WhyML, Why3 specification language, as an intermediate step between the EasyCrypt proof and OCaml code. In short, EasyCrypt proofs are first mapped into WhyML specifications following a purely syntactic translation process since the EasyCrypt functional language is very close to WhyML. Finally, we use Why3 powerful extraction mechanism to obtain executable OCaml code, a synthesis mechanism for which a semantic preservation proof exists [27]. Therefore, it is possible to deposit a high degree of trust in the final OCaml code.

Secure optimization of high-assurance cryptographic code: When developing a formal security proof inside EasyCrypt, it is natural to focus on reducing the proof effort, instead of focusing on the efficiency of specification. This typically leads to formalization for which the proof effort is reduced to the minimum, but where the code obtained will be slow, specially comparing to non-verified implementations. To overcome this drawback, we construct new methodologies and proof approaches that can be used to optimize EasyCrypt specifications.

The most challenging optimization strategy that we formalized is based on parallel computing, where we constructed a generic and modular *map-reduce* like EasyCrypt framework that works for an arbitrary number of parallel cores and for which we demonstrate that functional correctness and security properties of the sequential execution are preserved in the parallel model. This framework is of independent interest and can be re-used in the context of other existing and new EasyCrypt formalizations of cryptographic algorithms. Specific to our LPZK solution, we obtain a parallel implementation where a circuit is split into smaller independent circuits, and where the outputs of the cores are aggregated in a way that generates the same messages as the sequential execution.

Additionally, we also show how to replace data structures used to store private inputs and randomness with the goal of speeding up memory accesses. We focus on showing how to change a list-based implementation to an array-based implementation, which results in significant performance gains on the final implementation. Informally, our result states that, if the two implementations compute a message with the same type (or data structure), then this optimization can be securely done with the overhead of proving an isomorphism between the list and array storage, and that memory accesses will be *safe*, i.e., there will be no accesses to memory that is not allocated.

Limitations The Trusted Code Base (TCB) of our verified LPZK implementation includes EasyCrypt, the extraction tool and the OCaml compiler, as well as unverified OCaml libraries for multi-precision integers, for array operations, for the serialization of network messages and for the spawning of parallel cores. It is possible to reduce the TCB by relying on Jasmin [2] low-level verified arithmetic libraries [3]. In this case, a (manual) C code layer connects the extracted OCaml code to the Jasmin code, as further discussed in Appendix C.

Benchmarking Methodology Throughout the paper, we report multiple performance results for the different versions of our verified LPZK implementation. To avoid repeatedly describing the benchmarking environment, we briefly describe it here.

All measurements were conducted on an Amazon Web Services (AWS) EC2 instance, with an Intel Xeon Scalable processor clocked at 3.1 GHz, with 32 virtual cores (16 physical cores with two threads) and with 32 GB RAM capacity. We test our implementations against a matrix multiplication circuit, where we consider matrices of size 2^n, with incremental n. We also run our tests considering three different field sizes, namely $2^{30} - 2^{18} + 1$ (32 bit prime), $2^{61} - 1$ (64 bit prime) and $2^{255} - 19$ (256 bit prime), and use a 10 hours timeout.

For each test, we collect execution times of the prover and of the verifier, which include the time taken to compute the commit and to generate the commitment message (for the prover) and the time taken to parse the received message and decide the proof acceptance (for the verifier). From them, we derive the total protocol time and the *time-per-gate*. We measure the time-per-gate considering only the number of multiplication gates, which we obtain by 2^{n^3}.

Access to Our Development Our EasyCrypt formalization, proofs, and extracted executable software can be found at https://github.com/SRI-CSL/high-assurance-crypto/tree/main/high-assurance-zk/lpzk.

Paper Outline The rest of the paper is organized as follows. We give an overview of LPZK in Section 2. We then show how we developed our EasyCrypt formalization of LPZK and achieve the first version of our LPZK verified implementation in Section 3, including some performance measurements. In Sections 4 and 5, we present the optimizations that were made to the *reference* LPZK implementation, analyzing the performance gains of each optimization. We conclude the paper by discussing relevant related work in Section 6 before finishing up by pointing some conclusions and possible future research directions in Section 7.

2 Line-Point Zero Knowledge protocol

The LPZK protocol has been given two (different) presentations in [19] and [18]. We refer the reader to both of those presentations, and give a third, rather different presentation to simplify the illustration. We begin with a discussion of our choice of LPZK protocol, before describing the protocol at a high-level, and then give more low-level details that are crucial to understanding its formal verification.

The "Line-Point Zero Knowledge" protocol is actually a family of protocols, representing a series of optimizations and variations. In the language of [18], we call these protocols IT-LPZKv1, IT-LPZKv2, ROM-LPZKv1, and ROM-LPZKv2. The IT-LPZK protocols are information-theoretically secure in the random VOLE (rVOLE) model, while the ROM-LPZK protocols are secure in the random VOLE-random oracle (rVOLE-ROM) hybrid model. The v1 protocols are the baseline constructions, while the v2 constructions are the constructions which improve the communication cost either by generating more complex correlated randomness or by transferring communication to an offline step. Each of the IT-LPZK constructions require two times the communication cost of the corresponding ROM-LPZK protocols (when ROM-LPZKv2 is performed on a layered circuit).

In this work, we implement and formally verify IT-LPZKv1, although we include one technical modification from IT-LPZKv2 that reduces the computation cost. Namely, in the VOLE expression $\mathbf{v} := \mathbf{a}\alpha + \mathbf{b}$, we use the vector $\mathbf{b}$ rather than the vector $\mathbf{a}$ to store wire values. Our choice of IT-LPZKv1 was motivated by the desire to formally verify a VOLE-based NIZK protocol in an *as general as possible* setting, that does not require the assumption of the Random Oracle Model (ROM). We remark that IT-LPZKv1 is a reasonable choice in real-world settings: without a random oracle, the computation cost goes down, so IT-LPZKv1 and IT-LPZKv2 will outperform ROM-LPZKv1 and ROM-LPZKv2 over sufficiently fast networks, and, although the v2 protocols move some online work to the offline step, the v1 protocols are simpler and do not require any attention to be paid to circuit structure.

In what follows this section, we will first provide a theoretic overview of LPZK, before finishing with a gate-by-gate explanation of the IT-LPZKv1 protocol.

2.1 Theoretic LPZK overview

LPZK is a protocol whose goal is to prove satisfiability of arithmetic circuits in zero-knowledge. Without loss of generality, we will only consider arithmetic circuits that evaluate to 0. In LPZK, we treat the expression $\mathbf{v} = \mathbf{a}\alpha + \mathbf{b}$ held by the verifier as the evaluation of a line on a point α, and the expression held by the prover as a formal polynomial representation of that line $\mathbf{a}\alpha + \mathbf{b}$. We embed circuit values into the coefficients of that line, and then generate from here a collection of matching polynomial representations, where the prover holds the formal expression and the verifier holds the evaluation. We perform a series of manipulations so that the list of all coefficients in these selected polynomials is a small fraction of the circuit size (in fact, equal to $1/t|C|$, for a batching parameter t), and so that revealing these coefficients gives the verifier no information. Then, the prover reveals all coefficients from the selected expressions to the verifier. This gives total communication per multiplication gate equal to $2 + \frac{1}{t}$ and soundness error is equal to $t/|\mathbb{F}|$. Note that, in the EasyCrypt implementation, we take $t = 1$ for simplicity. The theoretic outline of LPZK is as follows:

1. Encode each wire value w corresponding to an input to the circuit or an intermediate gate value as the value b of a VOLE entry $v = a\alpha + b$. Generate a random VOLE $\mathbf{v} := \mathbf{a}\alpha + \mathbf{b}$ and have the prover send the verifier the vector $\mathbf{w} - \mathbf{b}$.

2. Prover and verifier generate matching quadratic expressions for each multiplication gate of the form $v_i v_j - v_k$, where the prover holds expressions over a formal variable t, and, if the prover is honest, the verifier holds the same expressions evaluated at $t = \alpha$. These quadratics will have zero constant term when the prover is honest, since $w_i w_j - w_k = 0$, and if the prover cheats or the witness fails, the quadratic will have nonzero constant term with overwhelming probability.

3. Prover and verifier refine the matching quadratic expressions by consuming another entry of VOLE. After "masking" the quadratic expression with fresh randomness, the prover can open the quadratic coefficient to the verifier, so that they now both hold matching linear expressions with zero constant coefficient if and only if $\mathcal{P}$ is honest.

4. $\mathcal{P}$ convinces $\mathcal{V}$ they hold matching linear expressions by batching over size t. If $\mathcal{P}$ is honest, $\mathcal{P}$ holds linear polynomials with zero constant coefficient, so the product of t such terms is a

polynomial of the form $c\alpha^t$, where c is a product of the corresponding coefficients. $\mathcal{P}$ sends this value c to $\mathcal{V}$, who checks it against $c\alpha^t$.

2.2 Gate-by-gate IT-LPZKv1 explanation

We present IT-LPZKv1 with reference to global vectors known or constructed by a prover ($\mathcal{P}$), a verifier ($\mathcal{V}$), and a dealer ($\mathcal{D}$). The dealer is an ideal functionality replaced by a VOLE protocol in the actual execution of the protocol. All parties have reference to a circuit C, which we here write as $C = (n, \mathcal{G})$, with n the total number of wires, and where each gate $g \in \mathcal{G}$ is of the form $(\oplus, i, j, k)$, with $\oplus \in \{+, \times\}$, with input wires of index i and j and an output wire at index k. A pseudo-code description of the protocol is depicted in Figure 2.

```
Dealer algorithm: $α, u = (a, b, a', b'); y = (v, v'), where
  Input wires: u = ($a, $b, ⊥, ⊥); y = (a α + b, ⊥)
  Add   wires: u = (a_l + a_r, b_l + b_r, ⊥, ⊥); y = (a α + b, ⊥)
  Mul   wires: u = ($a, $b, $a', $b'); y = (a α + b, a' α + b')

Prover algorithm: z = (m, m', c), where
  Input wires: z = (w-b, ⊥, ⊥)
  Add   wires: z = (⊥, ⊥, ⊥)
  Mul   wires: z = (w-b, a_l*a_r-a', a_l*w_r+a_r*w_l-a-b')
Final message: (z, a)

Verifier algorithm: f = (e, e', e''), where
  Input wires: f = (v+m, ⊥, ⊥)
  Add   wires: f = (e_l + e_r, ⊥, ⊥)
  Mul   wires: f = (v+m, v' + α*m', e_l*e_r-e-α*e')
Check: Output wire: e == n*α + w
       Batched checks: e_i'' == c_i*α
```

Figure 2: IT-LPZKv1 protocol, including descriptions of the dealer, prover, and verifier

The dealer, upon receiving the statement encoded as a circuit C, randomly samples a value α and generates vectors $\mathbf{a}, \mathbf{b}, \mathbf{v}, \mathbf{a}', \mathbf{b}', \mathbf{v}'$, each of length $|C|$, as follows. For input wires i, we have (a_i, b_i), generated randomly, $v_i = a_i\alpha + b_i$, and $a'_i = b'_i = v'_i = \bot$. For each addition gate, we have $(a_k, b_k, v_k) = (a_i + a_j, b_i + b_j, v_i + v_j)$, with $a'_i, b'_i, v'_i = \bot$. For each multiplication gate, we generate a_k, b_k, a'_k, b'_k randomly with $v_k = a_k\alpha + b_k$ and $v'_k = a'_k\alpha + b'_k$.

The prover, holding the secret witness w and all intermediate gate values (recall that the prover can evaluate the circuit on the clear), receives vectors $\mathbf{a}$, $\mathbf{a}$', $\mathbf{b}$, $\mathbf{b}'$ from the dealer, and computes the vectors $\mathbf{m}, \mathbf{m}'$, and $\mathbf{c}$ as follows. For each input wire, we have $m_i = w_i - b_i$, $m'_i = \bot$, $c_i = \bot$. For each addition gate, we have $(m_k, m'_k, c_k) = (\bot, \bot, \bot)$. For each multiplication gate, we have

$$(m_k, m'_k, c_k) = (w_k - b_k, a_i a_j - a'_k, a_i w_j + a_j w_i - a_k - b'_k).$$

Finally, the prover sends the vectors $\mathbf{m}, \mathbf{m}'$ to the verifier along with the vector $\mathbf{n} = (n_1, n_2, \dots)$, with $n_1 = a_n$ and for $i > 1$ we have $n_i = \prod c_j$, where the c_i's are split into batches of size t and multiplied together batch-by-batch.

The verifier receives a value α and a vector $\mathbf{v}$ from the dealer, and knows the desired output wire w_n. This value can be fixed to zero if one considers, without loss of generality, arithmetic circuits that evaluate to zero. It then computes the vectors $\mathbf{e}, \mathbf{e}', \mathbf{e}''$ as follows. For each input wire, we have $(e_i, e'_i, e''_i) = (v_i + m_i, \perp, \perp)$. For each addition gate, we have $(e_k, e'_k, e''_k) = (e_i + e_j, \perp, \perp)$. For each multiplication gate, we have

$$(e_k, e'_k, e''_k) = (v_k + m_k, v'_k + \alpha m'_k, e_i e_j - e_k - \alpha e'_k).$$

Finally, the verifier checks that $e_n = n_1 \alpha + w_n$ and that $e''_i = \alpha \cdot n_i$.

3 Verified LPZK implementation

In this section, we provide the details of our LPZK EasyCrypt formalization, highlighting some of its features with EasyCrypt code snippets. Our work leverages on previous IKOS formalization given in [6], out of which we re-used all ZK security definitions and protocol syntax, adapted to the DVNIZK model.

DVNIZK protocol syntax The starting point of our EasyCrypt formalization is the syntax definition of a DVNIZK protocol, given in Figure 3. Following the same approach as [6], this syntax definition makes a mix-use of abstract and concrete types and operators. Undefined data types and operators must be specified by each protocol at the instantiation step, including the witness, statement and randomness types. The output types of both the prover and verifier are hardwired into the formalization as a singleton and boolean value, respectively, since these will be the same for each possible instantiation of a DVNIZK protocol.

```
op relation : witness_t → statement_t → bool.
op language(x : statement_t) = ∃ w, relation w x.
type prover_input_t = witness_t * statement_t.
type verifier_input_t = statement_t.
op valid_rand_verifier :
        prover_rand_t → verifier_rand_t → verifier_input_t → bool.
type prover_output_t = unit.
type verifier_output_t = bool.
op commit : prover_rand_t → prover_input_t → commitment_t.
op prove : verifier_rand_t → verifier_input_t → commitment_t → bool.
type trace_t = commitment_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 c = commit r_p x_p in
  let b = prove r_v x_v c in (c, ((),b)).
```

Figure 3: DVNIZK protocol syntax

Because our formalization focuses on non-interactive protocols, we only specify one prover operator - `commit` - and one verifier operator - `prove`. Informally, the prover will run `commit`, producing a proof message that is sent to the verifier, that can then finish the ZK protocol by attesting the validity of the statement invoking `prove`. The honest protocol execution is specified by the `protocol` method. Both operators are abstract, and need to be realized when instantiating a concrete DVNIZK protocol. Moreover, both operators are de-randomized, meaning that randomness needs to be explicitly given to both the prover and the verifier. This is natural restriction, that essentially assumes that randomness is sampled uniformly by some honest random generator procedure, which, in the case of LPZK, is represented by the dealer functionality.

We would like to highlight the `valid_rand_verifier` operator. The goal of this operator is to assure that the randomness given to the verifier is correlated to the randomness given to the prover, and represents an essential component of our formalization, since the two parties will only be able to correctly execute the LPZK protocol if both randomness are correlated. Nevertheless, we do not provide a concrete specification of this predicate, since different designated verifier protocols can have different randomness correlated assumptions.

Security properties We re-use the security definitions formalized in [6], with small modifications that adapt them to capture the network model of DVNIZK protocols. Due to the similarities between the security definitions, we omit them in the main body of the paper and refer the reader to Appendix A for a more detailed description.

Arithmetic circuits We model arithmetic circuits (Figure 4) as a record with three elements. The first element is the topology of the circuit, that comprises the number of public input wires `npinputs`, the number of secret input wires `nsinputs`, the number of gates `ngates` and the number of output wires `noutputs`. The second element comprises the actual gates that form the circuit, modeled as an inductive tree, where the output of the circuit is given by the gate at the root of the tree, nodes correspond to arithmetic gates, and the input and constant gates form the leaves. In Figure 4 (and throughout the remaining of the paper), type `t` is used to refer to the type of elements of a finite field. Finally, the circuit description also encompasses the concrete definition of the output wires, represented by the `out_wires` record. This element is essential to our parallelization strategy, since it will facilitate the splitting of circuits by output wires.

```
type wire_t = t. type wid_t = int. type gid_t = int.
type gates_t = [
  | PInput of wid_t
  | SInput of wid_t
  | Constant of gid_t & t
  | Addition of gid_t & gates_t & gates_t
  | Multiplication of gid_t & gates_t & gates_t ].
type topology_t = { npinputs : int ; nsinputs : int ; ngates : int ; noutputs : int }.
type circuit_t = { topo : topology_t ; gates : gates_t ; out_wires : wid_t list }
```

Figure 4: Arithmetic circuits modeled as inductive tree

LPZK specification We use the same terminology followed in Section 2 to define the prover and verifier randomness required to perform a correct LPZK evaluation, as if it was provided by an honest dealer. The EasyCrypt randomness specification is portrayed in Figure 5. Succinctly, the prover will have a list composed of four field elements (`a`, `b`, `a’` and `b’`) per gate and the verifier will hold the random field element `alpha` and a list of random elements that are correlated to the prover randomness, i.e., `v = a * alpha + b` and `v’ = a’ * alpha + b’`. This property is attested by the `valid_rand_verifier` predicate. We also refer the reader to the `valid_rand_prover` predicate, where we define what constitutes valid prover randomness: we require all `a` values corresponding to the circuit gates to be different from zero. As it is going to be explained more ahead in this section, this restriction was made to simplify the soundness result.

We now show how the LPZK prover and verifier were formalized in EasyCrypt. Again, we follow the same nomenclature of Section 2 and the EasyCrypt definitions we present here can easily be matched against those given in Section 2.

We model the commitment message as a tree, following the same format used for the definition of arithmetic circuits. For each circuit gate, the prover commits values `m`, `m’` and `c`, captured

```
type ui_t = { a : t ; b : t ; a' : t ; b' : t }. type prover_rand_t = ui_t list.
op valid_rand_prover (r : prover_rand_t) (x : prover_input_t) : bool =
  let (w, st) = x in
  let (c, inst) = st in
  size r = c.‘topo.‘nsinputs + c.‘topo.‘npinputs + c.‘topo.‘ngates + 2 ∧
  (∀  k, 0 ≤ k < size r ⇒ (nth def_ui r k).‘a ≠ fzero).
type yi_t = { v : t ; v' : t }.
type verifier_rand_t = { alpha : t ; y : yi_t list }.
op valid_rand_verifier (rp : prover_rand_t) (rv : verifier_rand_t) (x : verifier_input_t) : bool =
  size rv.‘y = size rp ∧
  ∀ k, 0 ≤ k < size rv.‘y ⇒
    (nth def_yi rv.‘y k).‘v = (nth def_ui rp k).‘a*rv.‘alpha+(nth def_ui rp k).‘b ∧
    (nth def_yi rv.‘y k).‘v' = (nth def_ui rp k).‘a'*rv.‘alpha+(nth def_ui rp k).‘b'.
```

Figure 5: Prover and verifier randomness

by the `zi_t` record type. The final commitment message is comprised of the `z_t` tree structure (corresponding to **z** in Section 2) and of a finite field value (corresponding to **a** in Section 2).

```
type zi_t = { m : t ; m' : t ; c : t }.
type z_t = [
  | PInputZ of wid_t & zi_t
  | SInputZ of wid_t & zi_t
  | ConstantZ of gid_t & zi_t
  | AdditionZ of gid_t & zi_t & z_t & z_t
  | MultiplicationZ of gid_t & zi_t & z_t & z_t ].
type commitment_t = z_t * t.
```

Figure 6: Commitment message data type

The prover commits to the statement following the EasyCrypt specification illustrated in Figure 7. To improve readability, we focus only on the functionality of certain gates. Informally, the commitment message is built based on two operators: `gen_z` - transverses the circuit and produces the three commitment values `m`, `c` and `c'` for each gate - and `get_a` - outputs the random `a` value for the output gate of the circuit.

Before starting the commitment computation, the prover first invokes a pre-processing function dubbed `add_final_mul`. This function forces all circuits to end in a multiplication gate by adding a final multiplication gate to the root of the circuit tree. This small pre-processing layer was added to simplify the soundness proof, as is going to be explained more ahead in this section.

Upon receiving the commitment message, the verifier will build the data structure `f` (Figure 8), encompassing three field values per gates: `e`, `e'` and `e''`. This data structure also follows the same tree format as the circuit and commitment.

To validate a proof, the verifier first checks that the message it received is consistent with the original circuit (which is also checked for its validity), meaning that it will check if the prover produced a commitment for the actual circuit being evaluated. If this is the case, then the verifier will perform the multiplication gate checks corresponding to the second verifier check of Figure 2. Finally, the verifier computes `f` in order to obtain the value `e` for the output gate using the operator `get_e`, and check if `e` it is equal to $n \cdot \alpha$. The LPZK verifier formalization is portrayed in Figure 9.

Completeness Our completeness theorem states that LPZK achieves perfect completeness, assuming that the inputs are well formed and that the randomness of the two parties is correlated according to the predicate given in Figure 5. For the completeness theorem, we assume the ex-

```
op gen_z (u : prover_rand_t) (gg : gates_t) (xp : t list) (xs : t list) : z_t =
  with gg = PInput wid ⇒
    let b = (nth def_ui u wid).‘b in
    let w = eval_gates gg xp xs in
    PInputZ wid {| m = w - b ; m’ = 0 ; c = 0 |}
  ...
  with gg = Multiplication gid l r ⇒
    let wl = eval_gates l xp xs in let wr = eval_gates r xp xs in
    let w = wl * wr in
    ...
    MultiplicationZ gid {| m = w - b ; m’ = (al * ar) - a’ ;
                        c = ((al * wr) + (ar * wl)) - a) - b’ |}
                     (gen_z u l xp xs) (gen_z u r xp xs).
op commit (r : prover_rand_t) (x : prover_input_t) : commitment_t =
  let (w, st) = x in let (c, inst) = st in
  let c = add_final_mul c in
  (gen_z r c.‘gates inst w, get_a r c.‘gates).
```

Figure 7: Prover execution

```
type fi_t = { e : t ; e’ : t ; e’’ : t }.
type f_t = [
  | PInputF of fi_t
  | SInputF of fi_t
  | ConstantF of fi_t
  | AdditionF of fi_t & f_t & f_t
  | MultiplicationF of fi_t & f_t & f_t ].
```

Figure 8: Verifier data structure

istence of an honest dealer that provides correlated randomness to the prover and to the verifier. Formally, our EasyCrypt result is as follows.

Theorem 3.1 (Completeness (EasyCrypt))**.** *For all honest dealers* `D` *that produce correlated randomness, and for all valid witness* `w` *and statement* `x`*, if* $R(x, w)$ *holds then*

   Pr[Completeness(D).main(w,x)@ &m : res] = 1%r,

where `res` *is the output of the* `Completeness` *game.*

Proof (intuition). The proof is done by induction on the circuit, where, for every gate, we prove that, assuming that both parties hold correlated randomness, the prover will correctly compute the protocol relation and that verifier can successfully attest the proof. All proofs are reduced to simple field arithmetic properties, such as commutative, associative and distributive properties of addition and multiplication. □

Soundness The theoretic soundness error of LPZK is $\frac{1}{|\mathbb{F}|}$, where $\mathbb{F}$ is the size of the underlying finite field when we set the batching parameter to $t = 1$. In contrast with the completeness theorem, it is assumed that the prover randomness has been previously sampled and that the dealer will only be responsible to produce the appropriate verifier randomness. This result is represented in Theorem 3.2. Note that, in accordance with the `valid_rand_prover` of Figure 5, we are restricting the values of the prover random `a` values to be different than zero. If that was the case, then the soundness lemma of Theorem 3.2 would not hold, and we would need to account for a soundness error of $\frac{1}{|\mathbb{F}|-1}$. Restricting the value of the random `a` values to be different than zero simplifies the soundness result bellow.

```
op batch_check (f : f_t) (z : z_t) (alpha : t) : bool =
  ...
  with f = MultiplicationF fi fl fr ⇒
    if (is_multiplicationz z) then
      fi.`e'' = alpha * (as_multiplicationz z).`2.`c ∧
      batch_check fl (as_multiplicationz z).`3 alpha ∧
      batch_check fr (as_multiplicationz z).`4 alpha
    else false.
op gen_f (r : verifier_rand_t) (z : z_t) =
  with z = PInputZ wid zi ⇒
    let m = zi.`m in
    let v = (nth def_yi r.`y wid).`v in
    PInputF {| e = v + m ; e' = 0 ; e'' = 0 |}
  ...
  with z = MultiplicationZ gid zi zl zr ⇒
    let fl = gen_f r zl in let fr = gen_f r zr in
    let m = zi.`m in let m' = zi.`m' in
    let y = nth def_yi r.`y gid in
    ...
    let e = v + m in let e' = v' + (r.`alpha * m') in
    MultiplicationF {| e = e ; e' = e' ;
                       e'' = ((el * er) - e) - (r.`alpha * e') |} fl fr.
op prove (r : verifier_rand_t) (x : verifier_input_t) (c : commitment_t) : bool =
  let (z, b) = c in
  let (circ, inst) = x in
  if (valid_circuit circ) then
    let circ = add_final_mul circ in
    if valid_z z circ ∧ n ≠ fzero then
      let f = gen_f r z in
      if (batch_check f z r.`alpha) then
        get_e f = n * r.`alpha
      else false
    else false
  else false.
```

Figure 9: Verifier execution

Theorem 3.2 (Soundness (EasyCrypt))**.** *For all honest dealers* `D` *that produce correlated randomness against prover randomness* `rp`*, for all malicious provers* `MP` *and for all statements* `x`*, if the statement is not in the protocol language, then*

`Pr [ Soundness(D, MP).main(rp, x)@ &m : res ] ≤ 1%r / q%r`

where `q` *is the size of the finite field and* `res` *is the output of the* `Soundness` *game.*

Proof (intuition). To obtain the soundness result for the LPZK protocol, we require the circuit statement to end in a multiplication gate. Again, this can be done without loss of generality, since to all circuits can be added a multiplication by one gate without altering the final output value of the circuit.

This pre-computation greatly simplified the soundness proof, since most of the proof effort is going to be concentrated in the final multiplication gate. By eliminating the case where the random value a hold by a malicious prover is zero (which can only happen with negligible probability), we are left with an equation to be solved over one witness `w` field value, allowing us to derive a soundness error of at most $\frac{1}{|\mathbb{F}|}$. □

Zero-knowledge For the zero-knowledge result, we define a simulator that will execute following the exact same steps as the prover but, because it has no access to the witness, assumes that all witness values are zero. The zero-knowledge result is obtained by establishing an isomorphism between the honest prover execution and the simulator, as described in Theorem 3.3. For this theorem, we make use of a concrete prover random generator RP, that samples four random field elements per gate.

Theorem 3.3 (Zero-knowledge (EasyCrypt))**.** *For all honest prover random generator* RP*, for all distinguishers* D*, for all malicious verifiers* MV *and for all witness* w *and statement* x*, we have*

```
Pr[ GameReal(D, RP, MV).main(w,x)@ &m : res ] =
Pr[ GameIdeal(D, RP, MV, Simulator).main(w,x)@ &m : res ]
```

where res *is the output of the* ZKGame *game.*

Proof (intuition). We first establish an isomorphism between an honest prover's execution and the simulator. This isomorphism states that if the random values held by the prover and simulator are uniformly sampled, then the honest prover execution (that uses the correct witness values) and the simulator (that assumes all witness values are zero) are indistinguishable. As a consequence, a malicious verifier will have no advantage against the zero knowledge game, since it will not be able to differentiate if the commitment message receive came from an honest prover run or from the simulator □

First optimization step: reducing extra circuit iterations Before extracting OCaml code from our EasyCrypt formalization, we perform a preliminary optimization step that reduces extra circuit iterations made by the verifier. Following the definition depicted in Figure 9, one can observe that the verifier iterates over the circuit to check that the commitment message is consistent with the circuit and then performs another circuit iteration to generate f. Naturally, this party can condense the two circuit iterations into a single one, where gen_f now checks the consistency of the commitment while generating f at the same time.

Completeness, soundness and zero knowledge for this new version of LPZK were derived by proving that both the original and new versions have the same observable behavior. Informally, we prove that the new prove operator will produce the same decision as the *reference* one on all possible inputs. This allowed us to prove that: i. the completeness game of the new LPZK version has the same probability of the completeness game of the *reference* LPZK version; ii. the soundness game of the new LPZK version has the same probability of the soundness game of the *reference* LPZK version; and iii. that the real zero knowledge game of the new LPZK version is indistinguishable from the real game of the *reference* LPZK version and, therefore, the same simulator can be used to prove zero knowledge.

3.1 Automated extraction of executable code

The verified implementation of LPZK is obtained from the EasyCrypt formalization by using the EasyCrypt synthesis tool developed in [20]. Briefly, this tool performs a complete translation of the EasyCrypt specification, stopping the extraction at the finite field arithmetic operations level. This pruning was purposely performed in order to allow multiple instantiations of these operations with different arithmetic libraries. For the context of this work, we use the default GMP arithmetic library. However, the extraction driver can be tweaked (without requiring any tool expertise) to map finite field operations to other arithmetic libraries.

Table 1 summarizes a preliminary performance analysis made to our high-assurance LPZK implementation, following the benchmark infrastructure described in Section 1. The reported timings seem poor, specially when comparing to the execution times of the unverified implementation in [18]. These differences are greatly exacerbated as one goes deeper into bigger instances of the matrix multiplication test family. For example, for the MM64 test, this version of the LPZK verified

	MM16				MM32				MM64				MM128			
	PRV	VER	TOT	TPG	PRV	VER	TOT	TPG	PRV	VER	TOT	TPG	PRV	VER	TOT	TPG
$2^{30} - 2^{18} + 1$	606	379	985	240	22814	17435	40250	1228	1035719	933133	1968852	7511	-	-	> 10 h	-
$2^{61} - 1$	604	380	984	240	22817	17392	40209	1227	1034646	931599	1966246	7501	-	-	> 10 h	-
$2^{255} - 19$	606	381	987	241	22926	17594	40520	1237	1035185	946349	1981534	7558	-	-	> 10 h	-
#gates	8960 (4096)				59378 (32768)				535766 (262144)				4243456 (2097152)			

Table 1: Performance analyses of the LPZK verified implementation. Each test instance is identified by *MMX*, with X being the size of the matrix side. *PRV* represents the prover total time, *VER* represents the verifier total time, *TOT* represents the total protocol time and *TPG* represents the time-per-gate with respect to multiplications. All times are given in milliseconds, except the TPG, which is given in microseconds. Due to space constrains, we ommit the prover and verifier times for MM128, and focus only on the TOT metric, given in hours. The last row of the table portrays the total gates of the matrix multiplication circuit, with the number multiplications being shown between parentheses.

implementation is 6 orders of magnitude slower both in total execution time and in time-per-gate measurements, considering the $2^{61} - 1$ field.

We also compare the obtained performance against the verified implementation of MPC-in-the-Head given in [6]. The benchmark performance of [6] only considers circuits of up to 10,000 gates, which is smaller than the maximum of our tests. Nevertheless, comparing against the MM16 test instance (which has an approximate number of gates), we can conclude that the verified implementation of LPZK out-performs the MPC-in-the-Head verified implementation by a factor of 12 for primes that fit into a 64-bit word, and by a factor 14 for the $2^{255} - 19$ prime. From an orders of magnitude point of view, our LPZK implementation is two-orders of magnitude faster comparing total times and one-order of magnitude faster comparing the time-per-gate metric.

Finally, we analyse the time-per-gate metric. It would be expected to observe a constant time-per-gate measurement, independently of the matrix size. However, that is not the case for the verified LPZK implementation we describe here, which suggests that our implementation is following a quadratic behaviour, in contrast to the expected linear behavior. The quadratic complexity is the result of the usage of lists to store input and randomness data, inducing linear access times throughout the implementation. This issue is going to be addressed later in this paper, particularly when we change to an array-based implementation, with constant access time.

4 Optimization based on execution model: parallelism

This section describes how we obtain a verified parallel implementation of the LPZK protocol. We start by describing how we formalized parallel execution in EasyCrypt and how we used it to build a generic parallelism-supporting library for EasyCrypt, that is of independent interest and that can be re-used for other protocols (not only for zero-knowledge protocols) to speed EasyCrypt formalizations of parallelized versions thereof. We then show how we instantiated the parallelism library with LPZK by splitting the circuit according to output wires. This parallelization strategy demonstrated to have good performance results for highly parallelizable circuits such as matrix multiplication. We conclude this section with a discussion of the performance gains obtained by our parallelism approach.

4.1 Parallel RAM EasyCrypt formalization

We model parallelism in EasyCrypt following the parallel RAM (PRAM) design of [21]. The authors propose the PRAM architecture as a way to capture the scenario where each step of a RAM machine can branch to multiple processes that have access to the same memory. Informally, PRAM can be seen as a collection of RAM machines that are executing asynchronously, and its workflow can be described as follows. First, the computation description is split into *smaller* independent descriptions, each one being given to a parallel core, specified as RAM machines. For example, if the computation is described by means of an arithmetic circuit, then the circuit is divided into smaller circuits, that can be aggregated back to the original circuit. Then, the collection of RAM machines (i.e., the parallel processes) execute asynchronously, each one with its own internal state but sharing common read-only memory. When all RAM machines end their execution, their outputs are collected and combined into a single output, as if it was produced sequentially.

Our EasyCrypt PRAM formalization was inspired by the EasyCrypt project of [7]. In this work, the authors explored the role of verified compilers in the context of MPC from two possible compilation dimensions: i. one that formalizes the distributive computation of MPC protocols described as programs; and ii. one where the program is compiled from a high-level language to a low-level language, while keeping the same security guarantees of the source program We leverage the formalization given by the first compilation dimension in order to formalize the PRAM framework. Our secure parallel result is established following a UC-like approach, where an attacker will attempt to distinguish a *real* world from an *ideal* world, colluding with an adversarial environment that controls the flow of inputs and the collection of outputs to and from the protocol. In the *real* world, the attacker interacts directly with the parallel execution of the protocol. In the *ideal* world, the attacker interacts with a RAM machine that is sequentially computing the protocol. The parallel PRAM execution is secure if there exists a simulator that can emulate the real world view observable by the attacker, while interacting with the ideal functionality. This formalization choice was mainly motivated because it is a model that allowed us to reason not only about the equivalence of outputs, but also about the security of intermediate computations, capturing the desired twofold result: i. the asynchronous execution of the parallel RAM machines does not constitute a breach in the security of the protocol, or, in other words, the PRAM implementation computes the same thing as a sequential RAM machine even when the adversary controls the scheduler; and ii. the aggregation of the outputs of the PRAM cores is equivalent to a sequential execution.

Our formalization is parameterized by a *language* $\mathcal{L}$, that fixes how the computation is described. For the context of LPZK, $\mathcal{L}$ will correspond to arithmetic circuits, but we leave it abstract to increase the modularity of the parallel framework. This language will be used to characterize both the RAM and PRAM executions.

A RAM machine is modeled as a single party evaluating a circuit $\mathcal{C}$ written in the language $\mathcal{L}$. The evaluation is defined using a function `step` that is responsible to advance with the circuit evaluation, potentially modifying the internal state of the RAM machine. In the PRAM execution, a set of cores will be responsible to evaluate the original circuit $\mathcal{C}$. Each core locally executes its own *smaller* circuit, and the outputs are aggregated at the end. We consider two different execution configurations: one where cores advance asynchronously via their respective local `steps`, and another where cores advance synchronously at the same time.

The EasyCrypt formalization of both the RAM and PRAM semantics is portrayed in Figure 10, where `stepP` represents the asynchronous and `stepS` represents the synchronous core execution. In addition to the `step` procedures, the RAM and PRAM modules also disclose an initialization procedure, a procedure to provide inputs to the circuit and a procedure to collect output at the end of the evaluation. These interfaces fix the data types for the output of each parallel core (`execution_info_t`), possible circuit meta information that is required to reconstruct the original circuit (`meta_information_t`) and also the number of processors involved in the parallel evaluation. We enforce the distinction between `execution_info_t` and `output_t`: the former is used to capture the output of each parallel core, while the latter is used to capture the final output of the parallel evaluation, i.e., when all individual outputs are collected and aggregated.

```
module type RAM = {
  proc init(P : L) : unit
  proc step() : execution_info_t option
  proc setInput(x : input_t) : bool
  proc getOutput() : output_t option }.
```

```
module type PRAM = {
  proc init(meta : meta_information_t, Ps : L list) : unit
  proc stepP(id : processorId_t) : bool
  proc stepS() : execution_info_t option
  proc setInput(x : input_t) : bool
  proc getOutput() : output_t option }.
```

Figure 10: EasyCrypt RAM and PRAM computational models

The animation of a RAM or PRAM instance is done following a Universal Composability (UC) style approach, similar to the one formalized in [7]. Our framework is animated by two external entities: an *environment* $\mathcal{Z}$ and an *adversary* $\mathcal{A}$, that collude while interacting with the system. Looking at the interfaces of Figure 10, we observe that programs in both worlds progress based on `step` commands, which are delivered via the adversarial interface to $\mathcal{A}$ and to a simulator $\mathcal{S}$ (in the ideal world). In the real world (PRAM), the adversary can choose to either request a synchronous execution of all processes via the `stepS` interface or, alternatively, the adversary may drive a single process to progress in its local computations via `stepP`. In the ideal world, $\mathcal{A}$ has access to a single `step` method, that can possibly reveal some leakage. In either world, the environment can *activate* the adversary whenever it wishes to trigger the progress of the PRAM cores or of the sequential RAM machine. Additionally, the environment can also control the flow of inputs and outputs interacting with the RAM or PRAM semantics, through the procedures `setInput` and getOutput.

4.2 Generic *map-reduce* EasyCrypt library

The intuition behind our generic *map-reduce* formalization is the usage of multiple instances of RAM machines to capture the specification of the parallel cores. Indeed, PRAM will maintain a *parallel* state (`pstate_t`), consisting of a collection of the individual states of every RAM machine involved in the parallel computation. Our EasyCrypt *map-reduce* framework assumes that it will evaluate non-reactive functionalities, meaning that inputs can only be provided once at the beginning of the computation and outputs can only be collected once at the end of the computation.

We first show the formalization of a dedicated functional specification of a RAM machine in EasyCrypt, by instantiating the RAM interface of Figure 10 with the functional operators showed in Figure 11. The motivation behind the definition of the functional RAM model was to provide a more natural way to specify the execution of the parallel cores, since it allows us to use the functional operators to capture the workflow of the parallel cores.

The execution of a RAM machine is centered around its *state*, captured by the `state_t` type. At first, the empty state is initialized with a *circuit* (i.e., the computation description). Then, as the evaluation takes place, a RAM machine can be given input via `set_input` or, if the computation

```
type state_t.
op empty_state : state_t.
op init_state : P → state_t.
op set_input : state_t → input_t → state_t.
op step : state_t → state_t * execution_info_t option.
op get_input : state_t → input_t option.
op get_output : state_t → output_t option.
```

Figure 11: EasyCrypt functional RAM model

has ended, collected output from. The concrete circuit evaluation is captured by the `step` operator, that produces a new state and possible execution related information.

The realization of the parallel *map-reduce* execution on top of the PRAM model is depicted in Figure 12. Each processor (RAM machine) is initialized with its corresponding circuit. After, the execution can follow two different approaches. First, the cores can be executed one at a time using `stepP`. Because we only consider non-reactive functionalities, a successful core execution is only achieved if the core was already provided input and if it has no output (i.e., it has not reached the end of its circuit). And second, the entire set of cores can be animated at the same time using `stepS`. Processors share the same input values and output collection can only be done after all individual processes have finished their respective computation. Note that, following a UC-like formalization, it means that `stepP` and `stepS` can be invoked intertwined an arbitrary number of times, without a specific order.

```
module MapReduce = {
  proc init(meta_ : meta_information_t, Ps : L list) : unit = { ... }
  proc stepP(id : processorId_t) : bool = {
    st_i ← odflt empty_state (assoc pst id); r ← false;
    if (0 ≤ id < nprocesses ∧ has_input st_i ∧ !has_output st_i) {
      pst ← (id, fst (step st_i)) :: (assoc_rem id pst);
      r ← true;
    }
    return r;
  }
  proc stepS() : execution_info_t option = {
    while (i < nprocesses) {
      b <@ stepP(i);
      pst ← (i, odflt empty_state (assoc pst i)) :: (assoc_rem i pst);
    }
    y ← aggregate meta pst;
    return y;
  }
  proc setInput(x : input_t) : bool = { ... }
  proc getOutput() : output_t option = { ... } }.
```

Figure 12: Concrete PRAM formalization

The secure transformation of a sequential evaluation into a parallel one relies on the operators portrayed in Figure 13, which need to be provide for every instantiation of the parallel framework. The first one is the circuit split operator `split_circuit`, that takes as input a circuit and produces *smaller* independent circuits, together with all meta information required to reconstruct the original circuit. The next one is the circuit aggregation procedure `aggregate_circuit`. Intuitively, these two operators should cancel each other, in the sense that aggregating a circuit after splitting it should yield the same original circuit and vice-versa. Finally, our model specifies the `aggregate` function, that uses the circuit meta information and the parallel state where all cores have finished their execution in order to compile the outputs of all cores into a single output value, as if it was sequentially computed.

```
op split_circuit : P → meta_information_t * P list.
op aggregate_circuit : meta_information_t * P list → P.
op aggregate : meta_information_t → pstate_t → execution_info_t option.
```

Figure 13: Split and aggregate functionalities

Our model imposes the expected correctness assumptions regarding these functions. In more detail, we require that `aggregate`, using a parallel state that resulted from evaluating a set of circuits obtained via `split_circuit`, must produce the same output as a single RAM machine evaluating the original circuit. An instantiation of `aggregate` for which the above property hold is said to *correctly aggregate* a circuit. This property is the only property that users need to prove when using our parallelism library.

Our main parallel result is established by proving that splitting a circuit and evaluating it according to the PRAM description of Figure 12 is equivalent to evaluating a circuit sequentially using a dedicated RAM machine. In other words, we prove that a sequential execution of some algorithm can be securely transformed into a parallel execution, with the same observable behavior. Formally, we prove the following EasyCrypt theorem.

Theorem 4.1 (Secure parallel execution (EasyCrypt))**.** *For all environments $\mathcal{Z}$ and for all adversaries $\mathcal{A}$, for all circuits* `c` *and for all* `aggregate` *functions that correctly aggregate* `c`*, there exists a simulator* `S` *such that*

```
    Pr[ ParallelGame(Z, A).eval(c)@ &m : res ] =
    Pr[ SequentialGame(Z, A, S).eval(c)@ &m : res ]
```

where `ParallelGame` *is the execution of* `MapReduce` *from Figure 12,* `SequentialGame` *is the sequential RAM evaluation*

Proof (intuition). Intuitively, the goal of the simulator `S` is to intercept queries made by the adversary $\mathcal{A}$, and construct answers that will trick $\mathcal{A}$ into thinking it is dealing with a distributed program evaluation. In practice, it amounts to constructing a *simulated parallel execution*, that keeps in its internal state an emulation of the parallel semantics. Looking at the interface defined by the PRAM model, the parallel evaluation can easily be embedded in the simulator as it can store and manage local configurations for all existing parallel cores. Because the adversary does not get intermediate observations, we are to prove that the scheduler has no impact in the final result, a statement that follows from the independence of the computations. Note that, even though the different cores share the memory, they are not concurrently accessing the same memory locations.

The proof relies on simulator `S` presented above. It amounts to proving that the adversarial view in the PRAM model is indistinguishable from its view in the simulated RAM model. This is proven by keeping both the RAM and PRAM evaluation synchronized. When the `ParallelGame` invokes the `stepP` interface, the simulator replicates its behavior, which he can do since it maintains its local copy of the parallel execution. If, after executing the individual core, all cores have evaluated their respective circuit, the simulator will then proceed by evaluating the sequential RAM machine until it produces output. Similarly, when the `ParallelGame` invokes the `stepS` procedure, the simulator will also execute the entire set of cores at the same time, before executing the sequential RAM machine.

Finally, when $\mathcal{Z}$ requests output, the `ParallelGame` responds by aggregating the output of the parallel cores, whereas the simulator responds with the output of the sequential RAM machine. Because, by assumption `aggregate` correctly aggregates circuit `c`, both games will be equivalent. □

4.3 Verified parallel LPZK implementation

A verified parallel implementation of LPZK is obtained by instantiating the generic EasyCrypt parallelism with concrete LPZK operations as specified in Section 3. We formalize a parallel version of the LPZK prover and of the LPZK verifier, by first defining a sequential RAM wrapper for both entities and then by providing concrete realizations of the `split_circuit`, `aggregate_circuit` and `aggregate` operators. For the remaining of this section, we will focus on how we derived a parallel implementation of the LPZK prover. Achieving a parallel implementation of the LPZK verifier is analogous. To further attest the modularity of our parallelism framework, we discuss other instantiations of our EasyCrypt parallelism library in Appendix B.

The first step is defining a wrapper for the sequential LPZK specification detailed in Section 3 following the RAM syntax of Figure 11. The language considered is that of arithmetic circuits, as shown in Figure 4, where the witness, statement and randomness represent the circuit inputs, the output is the commitment message and the parallel core outputs (the `execution_info_t` data type) is the `z` structure.

The state of the RAM machine will store values that identify the circuit being evaluated, the witness, statement and randomness, and also of its final output. State modifications induced by the `init_state`, `set_input` and `step` functions are defined in the expected way. We illustrate the LPZK prover RAM wrapper in Figure 14. The final commitment construction is done when invoking the `get_output` method, whose responsibility is to take the output of the RAM machine and compute the field value `a`.

```
type state_t = { circ : L option ; w : witness_t option ; r : prover_rand_t option ; inst : instance_t option
     ; zo : execution_info_t option }.
op step (st : state_t) : state_t * execution_info_t option =
  if has_input st then
    let circ = oget (st.‘circ) in
    let zc = gen_z (oget st.‘r) circ.‘gates (oget st.‘inst) (oget st.‘w) in
    ({| circ = st.‘circ ; w = st.‘w ; r = st.‘r ; inst = st.‘inst ; zo = Some zc |}, Some zc)
  else (st, None).
op get_output (st : state_t) : output_t option =
  if st.‘zo ≠ None then
    let (z, c) = oget st.‘zo in
    let z’ = (get_a (oget st.‘r) (oget st.‘circ).‘gates, c) in
    Some (z, z’)
  else None.
```

Figure 14: LPZK prover RAM wrapper

Splitting is done by taking the description of the output wires exposed by the `output_wires` record of Figure 4 and extracting them from the gates record. The original circuit can be reconstructed back by aggregating the smaller circuits via addition gates. This aggregation strategy is correct because we are restricting the usage of arithmetic circuits that evaluate to zero and, therefore, the final circuit output can be obtained by adding the outputs of the smaller circuits. A similar approach is followed to combine the outputs of the parallel cores. Each processor ends its execution by outputting the `z` structure corresponding to the evaluated circuit, and the final value of `z` is derived by compiling all values of `z` using addition gates. We depict the EasyCrypt formalization of the `aggregate` function in Figure 15.

Executable OCaml code is obtained via code extraction, again using EasyCrypt to OCaml code generation tool of [20]. However, in contrast with the previous sequential LPZK implementation that we automatically synthesize using such tool, we were not able to perform a fully automatic code extraction from the EasyCrypt formalization, mostly because we are formalizing the execution

```
op aggregate (m : meta_information_t) (cs : pstate_t) : RAMProver.execution_info_t option =
  if has_output cs then
    let (topo, ys) = m in
    let c_0 = oget (oget (assoc cs 0)).‘RAMProver.zo in
    let id = topo.‘ngates - topo.‘noutputs in
    let reorganize = map (fun id ⇒ (id, oget (assoc cs id))) (iota_ 0 nprocesses) in
    let aggregation = foldl (fun st c ⇒ let (i, acc) = st in let (id, com) = c in (i+1, AdditionZ i {| m =
         fzero; m’ = fzero; c = fzero; |} acc (oget com.‘RAMProver.zo))) (id, c_0) reorganize in
    Some (snd aggregation)
  else None.
```

Figure 15: Aggregation of parallel cores outputs

of LPZK in a thread like environment, but we are leaving out of the formalization the operating system calls to generate the actual threads. These are part of our TCB and were implemented by resorting to the Domainslib OCaml library, a library that provides support for nested-parallel programming.

Table 2 summarizes the extraction methodology that we followed to obtain the parallel implementation from the proof, describing which components are automatically extracted and what needs to be manually implemented. We automatically extract the (verified) code that is going to be given to the different processors, how the original circuit is split and how the outputs of the processors are aggregated. The unverified part of the code is reduced to basic OCaml data types, the Zarith library to perform finite field arithmetic, and the Domainslib library used to create and launch threads.

	Content	Automatic	Manual
LPZK	Core code	All code	-
RAM LZPK	RAM wrapper	N/A	N/A
PRAM LZPK	PRAM implementation	`split_circuit` `aggregate`	Thread management

Table 2: Summary of the components that are automatically extracted and manually implemented

Table 3 gives a performance analysis of the PRAM LPZK implementation, realized following the same benchmarking approach of the sequential LPZK implementation. Our numbers reflect an execution with 4, 8, 16 and 32 parallel processes, taking advantage of the modularity of the parallelism framework. For this performance analysis, we focus only on the $\mathbb{F}_{2^{61}-1}$ field.

	MM16				MM32				MM64				M128			
	PRV	VER	TOT	TPG	PRV	VER	TOT	TPG	PRV	VER	TOT	TPG	PRV	VER	TOT	TPG
4 cores	242	116	358	87	9385	5622	15007	458	458383	325550	783933	2990	-	-	> 10 h	-
8 cores	122	73	195	48	4294	3352	7647	233	206723	197719	404442	1543	11954640	13342648	25297288	12063
16 cores	83	39	122	30	2353	1629	3983	122	105819	89089	194909	744	6064130	6680697	12744827	6077
32 cores	80	50	130	32	2200	1407	3607	110	87815	70990	158805	606	4597496	4851991	9449487	4506
#gates	8960 (4096)				59378 (32768)				535766 (262144)				4243456 (2097152)			

Table 3: Performance analysis of our parallel verified LPZK implementation, considering different instances of the MM test for the $\mathbb{F}_{2^{61}-1}$ field. All times are given in milliseconds, except the TPG, which is given in microseconds. The last row of the table portrays the total gates of the matrix multiplication circuit, with the number multiplications being shown between parentheses.

The performance benefits of introducing the parallel implementation are clear, and are particularly noticeable for bigger instances of the matrix multiplication problem. Concretely, even a 4 core setting is able to speed-up the LPZK evaluation time by a factor of 3, taking an order of magnitude off the time-per-gate on the MM16 and MM32 instances. Focusing on the bigger core setting of 32 cores, we observe 8-12x computation speed-up. In fact, considering a 32 core setting, our parallel LPZK formalization and extraction methodology leads to a 48% improvement on the total execution time of the protocol, while maintaining the high-assurance guarantees of the sequential LPZK implementation. Summing up, the total time of the protocol is reduced by a factor of 3 in a 4 core setting, by a factor of 5 in a 8 core setting, by a factor of 8-10 in a 16 core setting and by a factor of 8-12 in a 32 core setting.

Finally, we also note a significant reduction of the performance of the MM128 test. From more than 10 hours, our verified parallelism approach is able to reduce the execution time to 7 hours with 8 cores, to 3.5 hours with 16 cores and to 2.6 hours with 32 cores. Despite still being slow, it is a demonstration that our EasyCrypt parallelism framework can be used to reduce computation times whilst keeping the guarantees of sequential evaluations.

5 Optimization based on memory management: array-based implementation

In this section, we explore memory management performance gains by modifying the data structures used to store and access values in memory. Recalling the benchmark discussion of Section 3, our implementation observed a quadratic behavior, mostly because of all linear memory accesses that are performed due to the usage of lists. Therefore, we explored how changing to a an array-based constant access implementation could improve the performance of the verified LPZK implementation. At the end of the section, we will discuss the performance of two array-based implementations: i. a sequential one, that we use to compare against other manual, unverified implementations of NIZK protocols (including an unverified, highly optimized implementation of LPZK); and ii. a parallel one, that was obtained by taking advantage of the parallel EasyCrypt library described in Section 4, and that is the fastest implementation obtained by our methodologies

5.1 Array-based LPZK EasyCrypt formalization

The development of the new array-based LPZK implementation required the definition of an array EasyCrypt library, to be later used in the LPZK formalization. EasyCrypt already includes an array formalization as part of its standard library, however, array operations are specified over lists. For example, accessing an element of an array is defined by converting the array to a list and then accessing that element on the resulting list. This choice of formalization provides concrete realizations of array operations but does not remove the linear access time imposed by lists. Our (simple) array theory encompasses operators to create arrays, access array elements and converted between arrays and lists, as depicted in Figure 16. We purposely leave the array type and operations abstract, so that different array implementations can be used at the code extraction time.

```
type α array.
op size : α array → int.
op make : int → α → α array.
op get : α → α array → int → α .
op to_list : α array → α list.
op of_list : α list → α array.
```

Figure 16: EasyCrypt array library

		MM16				MM32				MM64				MM128			
		PRV	VER	TOT	TPG	PRV	VER	TOT	TPG	PRV	VER	TOT	TPG	PRV	VER	TOT	TPG
Sequential	$2^{30} - 2^{18} + 1$	3	4	7	1.7	26	28	54	1.7	236	268	504	1.9	1538	1496	3034	1.4
	$2^{61} - 1$	5	5	10	2.5	41	35	76	2.3	342	308	650	2.5	2194	1797	3991	2
	$2^{255} - 19$	19	18	37	9	156	148	304	9.2	1261	1169	2430	9.2	9985	10463	20449	9.7
32 cores	$2^{30} - 2^{18} + 1$	6	6	12	2.8	11	12	23	0.7	95	82	177	0.7	830	605	1435	0.7
	$2^{61} - 1$	7	6	13	3	15	13	28	0.8	123	94	216	0.8	982	690	1672	0.8
	$2^{255} - 19$	10	9	19	4.8	53	40	93	2.8	408	254	661	2.5	3498	2224	5722	2.7
	#gates	8960 (4096)				59378 (32768)				535766 (262144)				4243456 (2097152)			

Table 4: Performance analysis the array-based verified LPZK implementation. All times are given in milliseconds, except the TPG, which is given in microseconds. The last row of the table portrays the total gates of the matrix multiplication circuit, with the number multiplications being shown between parentheses.

The EasyCrypt formalization of the array-based LPZK is obtained by replacing all occurrences of the *list* type by the newly defined *array* type and all occurrences of `nth` (the list access EasyCrypt operator) by the `get` array operator. Although this optimization can be applied to any EasyCrypt formalization, we still do not have a generic, modular and automated way to easily apply it to any EasyCrypt code. Essentially, applying our memory management optimization approach involves manually changing the usage of lists to arrays, and then proving the equivalence between the two. This has to be done manually for every EasyCrypt formalization, unlike the parallelism optimization, which works as an automated *compiler* from sequential to parallel descriptions. Nevertheless, automating this process is something we envision as a potential future work direction.

The equivalence between the list-based and the array-based LPZK formalization is established by proving that the outputs of the prover and of the verifier are independent of the memory data structures being used. Concretely, this means proving that the prover produces the same commitment message given values stored using a list-based data structure or an array-based data structure and, likewise, proving that the verifier produced the same decision bit following either the list-based or array-based implementations. Note that we are not changing the type of the commitment message, i.e., both implementations are producing commitments with the same type (`type commitment_t = z_t * z’_t`). This allowed us to derive that: i. the completeness game of the array-based LPZK version has the same probability of the completeness game of the list-based LPZK version; ii. the soundness game of the array-based LPZK version has the same probability of the soundness game of the list-based LPZK version; and iii. that the real zero knowledge game of the array-based LPZK version is indistinguishable from the real game of the list-based LPZK version.

5.2 Verified array-based LPZK implementation

The array-based LPZK implementation was obtained by EasyCrypt code synthesis again relying on the tool of [20]. The developed EasyCrypt array library is instantiated at the code extraction level by plugging the standard OCaml array library directly into the tool as the backend for our EasyCrypt array library.

We conclude this section by reporting the performance of our verified array-based LPZK implementation, given in Table 4. We report the times of the sequential version and of the 32-core parallel version, that aggregates both the parallelism and array optimizations, giving the fastest times achieved by our verified LPZK implementation. We collect performance times over the three finite fields that we are considering.

	IT-LPZKv1				IT-LPZKv2				ROM-LPZKv2				VERIFIED IT-LPZKv1			
	PRV	VER	TOT	TPG	PRV	VER	TOT	TPG	PRV	VER	TOT	TPG	PRV	VER	TOT	TPG
opt. $2^{61} - 1$	23	16	39	0.04	18	12	30	0.03	234	213	447	0.4	-	-	-	-
gen. $2^{61} - 1$	1611	1120	2731	2.7	1304	900	2204	2.2	2340	1295	3635	3.6	1274	1051	2325	2.3
gen. $2^{255} - 19$	1976	1525	3501	3.5	1648	1176	2824	2.8	2172	1088	3260	3.3	4912	4472	9384	9.4

Table 5: Comparison between the high-assurance sequential array-based implementation of LPZK against the manual implementation of [18], as measured by executing a circuit with one million multiplication gates. The first row shows a comparison with the manual LPZK implementation using an optimized $\mathbb{F}_{2^{61}-1}$ backend (opt. $2^{61} - 1$), while the bottom rows show a comparison with the manual LPZK implementation using a generic NTL based field backend (gen. $2^{61} - 1$ and gen. $2^{255} - 19$). All times are given in milliseconds, except the TPG, which is given in microseconds

The performance gains are compelling, in all test instances and for all finite fields. Naturally, the most impressive gains are seen in the MM128 test instance, where the total evaluation decreased from more than 10 hours to just slightly over one and a half seconds ($\mathbb{F}_{2^{61}-1}$) or almost 6 seconds ($\mathbb{F}_{2^{255}-19}$), considering the 32 core parallel version. Indeed, changing to an array-based memory management reduces the overall complexity of the implementation, from a quadratic one to a linear one, as exposed by the constant time-per-gate measurement that is achieved in the array-based implementation. Consequently, the code execution times dropped considerably.

Comparison against unverified LPZK implementation For a fairer comparison, we will resort to the sequential array-based implementation when comparing to existing implementations of other ZK protocols, since most of them do not incorporate parallelism mechanisms. We start by giving a comparison with the the unverified implementation of LPZK given in [18], sketched in Table 5, where we compare the two implementations against a circuit with a million multiplication gates. For the manual LPZK implementation, we measured the times of IT-LPZKv1, IT-LPZKv2 and ROM-LPZKv2. The unverified LPZK implementation is instantiated first with a highly optimized backend for the $\mathbb{F}_{2^{61}-1}$ field and then with a generic finite field backend based on the NTL library, to provide a fair comparison with the verified LPZK implementation obtained from the EasyCrypt where field operations are done using OCaml bindings to the GMP library.

When comparing against the unverified LPZK implementation that uses the optimized $\mathbb{F}_{2^{61}-1}$ field, the verified LPZK implementation is 70x slower, which is reflected in the 2 orders of magnitude that separate the TPG measurement in both. This difference is amplified if one compares against the manual IT-LPZKv2 implementation. However, when the field operations are done using a generic field backend (which represents the fairest comparison between the verified an unverified implementations), we can conclude that our verified implementation is actually slightly faster that the unverified one. Concretely, we can observe 1.3x performance gains in the prover, verifier and total protocol times. Even considering IT-LPZKv2, the performance of the verified LPZK implementation achieves similar performance, even being slightly faster in terms of prover execution time. This contrasts with the performance measured for the $2^{255} - 19$ field, where the unverified LPZK implementation is actually 2.7x faster than the verified one. Since both implementations do not make algorithmic changes with respect to the finite field, we conjecture that these discrepancies come from differences in the arithmetic implementations made by the NTL and GMP libraries.

Comparison against unverified state-of-the-art NIZK protocols Taking advantage of Table 3 of [18], we can also compare our verified LPZK implementation against other state-of-the-art NIZK protocols and zkSNARKs, namely Groth16 [24], Virgo [33] and QuickSilver [31], that we summarize in Table 6. This table provides the execution times of the aforementioned protocols when

Groth16				Virgo				QuickSilver				Verified LPZK			
PRV	VER	TOT	TPG	PRV	VER	TOT	TPG	PRV	VER	TOT	TPG	PRV	VER	TOT	TPG
21k	2	21k	21	478	12	490	0.5	128	128	256	0.3	1274	1051	2325	2.3

Table 6: Comparison between the high-assurance sequential array-based implementation of LPZK against other state-of-the-art NIZK protocols, as measured by executing a circuit with one million multiplication gates. The protocols are all executed over $\mathbb{F}_{2^{61}-1}$. All times are given in milliseconds, except the TPG, which is given in microseconds

evaluating a circuit with one million gates over $\mathbb{F}_{2^{61}-1}$. All measurements were done on top of an AWS machine with equal specifications to the one used to collect the benchmarks of our verified LPZK implementation. The values reported put our sequential array-based LPZK implementation with better prover performance than Groth16, and with equal performance when comparing to Virgo. Nevertheless, those implementations achieve better verification times, also due to the nature of the protocols.

We also compare our code against other ZK codebases, based on MPC-in-the-Head, concretely that of ZKBoo [22], that also relies on parallelism. The authors report a 13ms prover time and a 5ms verifier time for the evaluation of a fine-tuned SHA-1 circuit with 40 AND gates, 372 XOR gates and 325 addition gates, that combines operations over $\mathbb{F}_{2^{32}}$ and over $\mathbb{F}_2$, using an 4 GHz 8 core machine. Because doing a single run of LPZK for a boolean circuit would yield a soundness error of $\frac{1}{2}$, LPZK requires the performance of a series of protocol repetitions to achieve a negligible soundness error. Concretely, to achieve a 2^{-80} soundness error as the experiences performed by ZKBoo, our implementation reports an execution time of 104ms for the prover and 96ms for the verifier, a 8x penalty compared with the time reported by ZKBoo. Note, nevertheless, that our benchmarking environment uses slower processing cores.

Finally, we refer the reader to the *32 core* line of Table 4, which portrays the fastest execution times achieved by our high-assurance LPZK implementation combining both optimization paths proposed in this paper. The purpose of this paper was to explore new techniques to close the performance gap between verified and unverified implementations of cryptographic mechanisms. In that sense, although it is an unfair comparison, such implementation is an important step in the direction of optimizing high-assurance cryptographic implementations, putting our final verified LPZK implementation in a very competitive position, even when compared with other unverified implementations of ZK protocols.

6 Related work

In this section, we discuss the state-of-the-art from the perspective of the recent accomplishments in VOLE-based NIZKs and from the perspective of the application of computer-aided cryptography to ZK protocols.

6.1 VOLE-based NIZKs

The original work [16] showing how two parties could efficiently generate vector oblivious linear evaluation (VOLE)-type correlated randomness, also observed that existing honest-verifier ZK linear interactive proof (HVZK-LIP) schemes could be compiled to NIZK over VOLE. Follow up work developed fast and efficient VOLE-based NIZKs. The first wave of these efforts included three works: LPZK [19], Mac'n'Cheese, [12], and Wolverine [29], where each used one entry of the VOLE to encode each gate of the circuit (treating the VOLE as a message authentication code, an

IT-MAC). LPZK achieved the best communication in the random oracle model for arithmetic circuits, requiring 1 field element of communication per gate, and gave the only construction that was information-theoretically secure in the random VOLE model, without requiring a random oracle. Wolverine required 2 elements of communication per gate, but could also treat boolean circuits, and Mac'n'Cheese applied stacked disjunction techniques to give more efficient constructions for special subclasses of circuits.

From here, improvements have continued developing further optimization, cross-polination, and diversification. Quicksilver [31], a follow-up work to Wolverine, combined techniques from Wolverine and LPZK to achieve one field element of communication per gate in the boolean setting, while additionally giving very small communication costs in the special case where a circuit could be represented as a low-degree polynomial. LPZKv2 [18] improved further to requiring $\frac{1}{2}$ elements of communication per gate in the case of *layered* circuits, using either more complex correlated randomness, or by moving half of the communication to an offline step. The Mac'n'Cheese protocol was extended to protocols Appenzeller and Brie [10] to handle switching between representations over different fields and rings, so that a circuit could be written with some gates over $\mathbb{F}_p$ and other gates over $\mathbb{F}_{2^k}$.

More recent work [30] presented a construction that requires $|C|^{3/4}$ communication in the general case at the cost of the use of an additively homomorphic encryption scheme (AHE), and Moz$\mathbb{Z}_{2^k}$arella [11] extended the VOLE-based ZK approach to the ring $\mathbb{Z}_{2^k}$.

6.2 Computer-aided cryptography applied to ZK protocols

The application of computer-aided cryptography mechanisms to ZK protocols has been the focus of recent research, the most closely related to our work being that of [6], where the authors formalized and automatically synthesized ZK protocols based on the IKOS framework [6]. Compared to previous work, the authors there consider 3-pass ZK protocols, and give both a machine-checked proof of security for IKOS and a formally verified implementation, including 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. Their formalization follows the original IKOS construction given in [25] and uses the standard syntax and security notions for ZK proofs, MPC protocols and commitment schemes. This has the advantage of allowing one to build on and to deploy standard components, but introduces the challenge of formalizing more complex security proofs.

Independently, the work of [28] presented a machine-checked security proof for a class of Σ-protocols that follows the approach to IKOS introduced by the ZKBoo protocol [22], which is an important optimized derivative of the MPC-in-the-Head 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 ZK. 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.

Prior to the above research, works in [9] and [17] were the first to formalize a special class of Σ^{ϕ}-protocols in CertiCrypt, a predecessor of EasyCrypt implemented as a Coq library, and CryptHOL, respectively. Both works proved the security of general AND and OR composability theorems for Σ^{ϕ}-protocols, formalized abstract and concrete commitment scheme primitives and proved a construction of commitment schemes from Σ-protocols.

Another computer-aided cryptography treatment of ZK protocol was the work of [4], where the authors developed a full-stack verified framework for developing ZK proofs. The proposed framework is composed of two compilers: i. a non-verified optimizing ZK compiler that translates high-level ZK proof goals to C or Java implementations; and ii. a verified compiler that generates a reference implementation The authors provided a CertiCrypt machine-checked proof attesting that the reference implementation satisfies the ZK properties and that the optimized implementation has the same observable behavior as the reference implementation, for any goal. This work leverages the results from [9], extended with AND compositions of Σ^{GSP}-protocols.

7 Conclusion and Future Work

In this paper, we provide an additional step in the effort of closing the performance gap between verified implementations and non-verified implementations of cryptographic primitives, using the LPZK protocol as a case-study. Our approach to securely generate an optimized high-assurance implementation of LPZK was mechanically verified in EasyCrypt and encompasses a generic and modular parallelism framework, tolerating an arbitrary number of cores, and a memory-based optimization path from lists to arrays.

The effects of the optimization methodologies we propose are further demonstrated by the graph depicted in Figure 17, where we show the averages of the time-per-gate measurements for the different versions of the implementation, starting from the list-based sequential implementation to the array-based 32 core parallel implementation. This graph shows how the verified optimizations were able to reduce the complexity of the implementation from quadratic to linear, ending up with a final codebase that achieves competitive performance even when compared with unverified implementations of other ZK protocols. The green line (concerning the sequential array-based implementation) and the dark blue line (concerning the 32-core parallel array-based implementation) are overlapped.

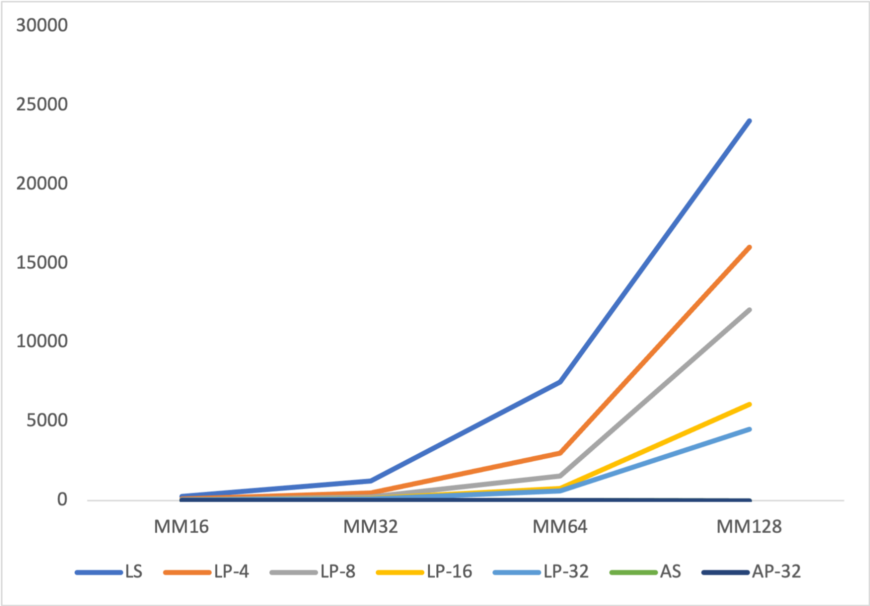

Figure 17: Time-per-gate evolution with respect to the matrix multiplication circuit. LS - list-based sequential implementation (Section 3) LPX - list-based parallel implementation (Section 4), with X denoting the number of cores AS - array-based sequential implementation (Section 5) AP-32 - array-base parallel implementation, using 32 cores

Acknowledgement

We would like to thank Sabine Oechsner and Peter Scholl for their valuable comments that helped us identifying a mistake concerning the batch multiplication checks made by the verifier. This error was present both in the original EasyCrypt proof and in the resuting OCaml implementation. It has been addressed and is now fixed.

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 the author(s) and do not necessarily reflect the views of the United States Government or DARPA.

References

[1] Jose Bacelar Almeida, Endre Bangerter, Manuel Barbosa, Stephan Krenn, Ahmad-Reza Sadeghi, and Thomas Schneider. A certifying compiler for zero-knowledge proofs of knowledge based on σ-protocols. Cryptology ePrint Archive, Paper 2010/339, 2010. https://eprint.iacr.org/2010/339.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1807–1823, 2017.

[3] 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. *CoRR*, abs/1904.04606, 2019.

[4] 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. Cryptology ePrint Archive, Paper 2012/258, 2012. https://eprint.iacr.org/2012/258.

[5] 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. Cryptology ePrint Archive, Paper 2017/821, 2017. https://eprint.iacr.org/2017/821.

[6] José Bacelar Almeida, Manuel Barbosa, Manuel L Correia, Karim Eldefrawy, Stéphane Graham-Lengrand, Hugo Pacheco, and Vitor Pereira. Machine-checked zkp for np-relations: Formally verified security proofs and implementations of mpc-in-the-head. Cryptology ePrint Archive, Paper 2021/1149, 2021. https://eprint.iacr.org/2021/1149.

[7] José Carlos Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Hugo Pacheco, Vitor Pereira, and Bernardo Portela. A formal treatment of the role of verified compilers in secure computation. *Journal of Logical and Algebraic Methods in Programming*, 125:100736, 2022.

[8] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. Cryptology ePrint Archive, Paper 2019/1393, 2019. https://eprint.iacr.org/2019/1393.

[9] 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, 2010.

[10] Carsten Baum, Lennart Braun, Alexander Munch-Hansen, Benoît Razet, and Peter Scholl. Appenzeller to brie: Efficient zero-knowledge proofs for mixed-mode arithmetic and z2k. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 192–211. ACM, 2021.

[11] Carsten Baum, Lennart Braun, Alexander Munch-Hansen, and Peter Scholl. Moz$\mathbb {Z}_{2^k}$arella: Efficient vector-ole and zero-knowledge proofs over $\mathbb {Z}_{2^k}$. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part IV*, volume 13510 of *Lecture Notes in Computer Science*, pages 329–358. Springer, 2022.

[12] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part IV*, volume 12828 of *Lecture Notes in Computer Science*, pages 92–122. Springer, 2021.

[13] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 784–796, 2012.

[14] 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.

[15] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K Rustan M Leino, Jacob R Lorch, Bryan Parno, Ashay Rane, Srinath TV Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In *USENIX Security Symposium*, volume 152, 2017.

[16] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. 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 896–912. ACM, 2018.

[17] David Butler, Andreas Lochbihler, David Aspinall, and Adria Gascon. Formalising σ-protocols and commitment schemes using crypthol. Cryptology ePrint Archive, Paper 2019/1185, 2019. https://eprint.iacr.org/2019/1185.

[18] Samuel Dittmer, Yuval Ishai, Steve Lu, and Rafail Ostrovsky. Improving line-point zero knowledge: Two multiplications for the price of one. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 829–841. ACM, 2022.

[19] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-point zero knowledge and its applications. Cryptology ePrint Archive, Paper 2020/1446, 2020. https://eprint.iacr.org/2020/1446.

[20] Karim Eldefrawy and Vitor Pereira. A high-assurance evaluator for machine-checked secure multiparty computation. Cryptology ePrint Archive, Paper 2019/922, 2019. https://eprint.iacr.org/2019/922.

[21] Naomi Ephraim, Cody Freitag, Ilan Komargodski, and Rafael Pass. Sparks: Succinct parallelizable arguments of knowledge. Cryptology ePrint Archive, Paper 2020/994, 2020. https://eprint.iacr.org/2020/994.

[22] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. Zkboo: Faster zero-knowledge for boolean circuits. Cryptology ePrint Archive, Paper 2016/163, 2016. https://eprint.iacr.org/2016/163.

[23] Shafi Goldwasser, Silvio Micali, and Chales Rackoff. The knowledge complexity of interactive proof-systems. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 203–225. 2019.

[24] Jens Groth. On the size of pairing-based non-interactive arguments. Cryptology ePrint Archive, Paper 2016/260, 2016. https://eprint.iacr.org/2016/260.

[25] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 21–30, 2007.

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

[27] Mário José Parreira Pereira. *Tools and Techniques for the Verification of Modular Stateful Code*. Theses, Université Paris Saclay (COmUE), December 2018.

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

[29] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1074–1091. IEEE, 2021.

[30] Chenkai Weng, Kang Yang, Zhaomin Yang, Xiang Xie, and Xiao Wang. Antman: Interactive zero-knowledge proofs with sublinear communication. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 2901–2914. ACM, 2022.

[31] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 2986–3001. ACM, 2021.

[32] Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.

[33] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. Cryptology ePrint Archive, Paper 2019/1482, 2019. https://eprint.iacr.org/2019/1482.

[34] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806, 2017.

A Designated verifier non-interactive ZK **EasyCrypt** security definitions

We re-use the security definitions formalized in [6], with small modifications that adapt them to capture the network model of DVNIZK protocols. For simplicity, we will only focus on the major differences between the two formalizations and refer the reader to [6] for a more detailed description of how ZK security properties can be formalized in EasyCrypt.

The main difference between the two models lies on the specification of the adversarial entities over which security definitions are quantified. For example, a malicious prover entity used to parameterize the soundness game is specified by the interface depicted in Figure 18, that discloses only one procedure, while the one of [6] discloses two procedures because IKOS is *commit-challenge-response* three pass protocol.

```
module type MProver_t = {
  proc commit(rp : prover_rand_t, x : statement_t) : commitment_t
}.
```

Figure 18: Malicious prover interface

This difference is also reflected in the soundness game, showed in Figure 19, where `RandV_t` models a verifier random generator procedure.

```
module Soundness(R : RandV_t, MP : MProver_t) = {
  proc main(rp : prover_rand_t, x : statement_t) : bool = {
    c <@ MP.commit(rp, x);
    rv <@ R.gen(rp);
    b ← prove rv x c;
    return b;
  }
}.
```

Figure 19: DVNIZK protocol soundness game

In what concerns the zero-knowledge property, we use the formalization of the *single-run zero knowledge* property of [6]. This definition is used as an intermediate step to obtain the standard zero-knowledge result for IKOS, but it is enough to capture standard zero-knowledge for LPZK. Zero-knowledge if formalized based on the *real* and *ideal* world experiences, that are specified by the `ZKGame` module, which, by itself, is parameterized by either a real-world evaluator or an ideal-world evaluator. The zero-knowledge formalization of DVNIZK protocols is represented in Figure 20, where `RandP_t` models a prover random generator procedure.

Both evaluators are defined according to the `Evaluator_t` interface. Because of space constrains, we omit the definition of the *real-world* evaluator, and focus only on the definition of the *ideal-world*, that we show in Figure 21.

The `IdealEvaluator` module has the responsibility of animating the interaction between a simulator and a malicious verifier. Intuitively, the goal of the simulator is to mimic the behavior of the prover and provide the malicious verifier with a commitment message that is indistinguishable from one generated by an honest prover execution, without knowing the secret witness. The execution is then shifted back to the `ZKGame`, where the trace produced by the evaluator is given to a module of the `Distinguisher_t` type, an entity that will try to differentiate between an honest

```
module type Distinguisher_t = {
  proc guess(_ : witness_t * statement_t * trace_t option) : bool
}.
module type Evaluator_t = {
  proc eval(w : witness_t, x : statement_t, rp : prover_rand_t) : trace_t option
}.
module ZKGame (D : Distinguisher_t) (RP : RandP_t) (E : Evaluator_t) = {
  proc main(w : witness_t, x : statement_t) : bool = {
    rp <@ RP.gen(w,x);
    ctr <@ E.eval(w,x,rp);
    b' <@ D.guess(w,x,ctr);
    return b';
  }
}.
```

Figure 20: DVNIZK protocol zero-knowledge game

```
module type MVerifier_t = {
  proc prove(x : statement_t, c : commitment_t) : bool
}.
module type Simulator_t = {
  proc gen_commitment(rp : prover_rand_t, x : statement_t) : commitment_t option
}.
module IdealEvaluator (MV : MVerifier_t) (S : Simulator_t) = {
  proc eval(w : witness_t, x : statement_t, rp : prover_rand_t) : trace_t option = {
    ret ← None;
    oc <@ S.gen_commitment(rp, x);
    if (oc ≠ None) {
      c ← oget oc;
      b <@ MV.prove(x, c);
      ret ← Some c;
    }
    return ret;
  }
}.
```

Figure 21: *Ideal-world* zero-knowledge evaluator

protocol execution (`RealEvaluator`) and a simulated one (`IdealEvaluator`). If the two evaluators are indistinguishable, it will output 1 with the same probability on both the *real* and *ideal* worlds.

B Other applications of the EasyCrypt parallelism library

In this appendix, we will explore the modularity of the EasyCrypt PRAM-based parallelism library by demonstrating how it can be instantiated with other protocols and how it can be used in application scenarios different than ZK protocols. Because our framework is best suited to work with circuit-based cryptographic primitives, we will focus on other EasyCrypt formalizations that also represent computations by means of arithmetic/boolean circuits. Concretely, we will discuss how to obtain secure parallel implementations following our PRAM model of:

1. the MPC-in-the-Head formalization of [6], showing how to split the relation circuit to speed-up the construction of the commitment message

2. the garbling scheme formalization of [5]. showing how the garbling scheme procedure can be parallelized to speed-up the overall computation

B.1 Parallel formalization of MPC-in-the-Head

The IKOS [25] construction is a ZK paradigm that combines a multiparty computation (MPC) protocol with a commitment scheme to yield a ZK protocol. Informally, the prover executes an MPC protocol *in its head*, i.e., it will emulate the interactions between parties, producing the communication trace of the MPC protocol. The prover commits to the traces (views) of each party and sends the corresponding commitments to the verifier. The verifier then challenges the prover by selecting a set of parties, of which the commitments are open by the prover. Finally, the verifier accepts the proof if the prover successfully opened the commitments to the views, if the deterministic output of those views is 1 (true) and if the views are consistent with each other, in the sense that the outgoing messages implicit in one views are identical to the incoming messages reported in another view.

The IKOS formalization of [6] is actually a twofold project. First, the authors formalized the IKOS modular construction, that was then instantiated with two MPC protocols: the BGW protocol [14] and Maurer's MPC protocol [26]. In this section, we will focus on the BGW-based IKOS instantiation, since it is the one that resembles our LPZK EasyCrypt specification the most.

Similarly to our formalization, the BGW-based IKOS instantiation also adopts a tree-like format, supporting the same arithmetic gates supported by LPZK. Therefore, an equal strategy can be employed in order to split the circuit being evaluated, by creating a series of independent *smaller* circuits, according to output wires. Each parallel core will then be responsible to produce the communication view that corresponds to the piece of circuit that was assigned to it. However, output aggregation is much simpler: because there is no communication on addition gates, aggregating the views produced by the parallel processes is simply a matter of concatenating the views together. The actual computation of the commitment message after outputs are aggregated is done by applying the commitment scheme to the re-constructed view. We provide on Figure 22 a sketch of a realization of our *map-reduce* EasyCrypt framework.

```
op split_circuit (c : L) : meta_information_t * L list =
  let ys = c.‘out_wires in
  let topo = c.‘topo in
  let gg = c.‘gates in
  ((topo, ys), map (fun y ⇒ {| topo = topo ;
                              gates = oget (get_gate gg y) ;
                              out_wires = [y] |}) ys).
op aggregate (m : meta_information_t) (cs : pstate_t) : RAMProver.execution_info_t option =
  if has_output cs then
    let (topo, ys) = m in
    let c_0 = oget (oget (assoc cs 0)).‘RAMProver.zo in
    let id = topo.‘ngates - topo.‘noutputs in
    let reorganize = map (fun id ⇒ (id, oget (assoc cs id))) (iota_ 0 nprocesses) in
    let aggregation = foldl (fun st c ⇒ let (id, v) = c in st ++ [v]) c_0 reorganize in
    Some aggregation
  else None.
```

Figure 22: IKOS instantiation of the EasyCrypt parallelism library

B.2 Parallel formalization of garbling schemes

Another interesting use case of our PRAM formalization resides on its application to legacy EasyCrypt formalizations, particularly that of [5], where the authors provide a high-assurance implementation of Yao's secure function evaluation (SFE) protocol [32], synthesized from an EasyCrypt formalization of the same protocol. The formalization closely follows the design given by Bellare, Hoang and Rogaway in [13], where the authors provide a provable-security treatment to garbling schemes, and show their applications as a building block to achieve SFE.

In [5], the authors adapt the same circuit syntax disclosed in [13]. Concretely, circuits are specified as a six-tuple $C = (n, m, q, A, B, G)$, where n is the number of input wires, m is the number of output wires and q is the number of gates. These three parameters are used to define a convention for the circuit wire organization as follows: input wires have IDs in $[1; n]$, gates have IDs in $[n + 1; n + q]$ and output wires have IDs in $[n + q - m + 1; n + q]$. A and B are functions that map the ID of a gate to its left and right incoming wires, respectively. Finally, G identifies the functionality of each gate.

Specifying a circuit splitting procedure for this circuit format is not as straightforward as in the previous example. One would need to write a function that, for all output wires, finds the part of the circuit that produces it. Aggregation can be done by taking the garbled circuits produced by the parallel cores and sorting them by wire IDs. These procedures are depicted in Figure 23.

```
op split_circuit (c : L) : meta_information_t * L list =
  let (n, m, q, A, B, G) = c in
  let ys = iota_ (n + q - m + 1) (n + q) in
  (((n, m, q, A, B), ys), map (fun y ⇒ (n, m, q, A, B, get_circuit y c)) ys).
op aggregate (m : meta_information_t) (cs : pstate_t) : RAMProver.execution_info_t option =
  if has_output cs then
    let (topo, ys) = m in
    let (n, m, q, A, B) = topo in
    let c_0 = oget (oget (assoc cs 0)).‘RAMProver.zo in
    let reorganize = map (fun id ⇒ (id, oget (assoc cs id))) (iota_ 0 nprocesses) in
    let aggregation = foldl (fun st c ⇒ let (id, v) = c in st ++ [v]) c_0 reorganize in
    Some (n, m, q, A, B, sort aggregation)
  else None.
```

Figure 23: Garbling scheme instantiation of the EasyCrypt parallelism library

C Reducing the TCB: Jasmin as the finite field arithmetic backend

A natural step forward after having a working verified implementation of LPZK is to try to reduce the TCB (i.e., replace unverified pieces of code with verified ones) without compromising on the efficiency of the overall implementation. To that end, we took advantage of the verified field arithmetic libraries written in Jasmin that were developed in the scope of the verified IKOS implementation developed in [6] and did a new code extraction from the EasyCrypt proof, this time pruning the resulting OCaml code at the field arithmetic level. These were specified to match the interface with the Jasmin code. The connection between OCaml and Jasmin is done via a thin layer of C code that was manually written, following the same approach of [6].

Table 7 reports the execution times obtained when the (unverified) Zarith library is replace with a (verified) Jasmin library for field arithmetic. Again, we are reporting the execution times obtained against the matrix multiplication test family, incrementing the matrix side. However, data from this tests 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.

Interestingly, introducing the Jasmin field arithmetic backend in replacement for Zarith does not compromise the overall efficiency of the protocol. In fact, Table 7 actually shows slight improvements. Nevertheless, the benefits of using the Jasmin backend for field arithmetic are not strictly related to efficiency: because the field operations are verified, we are actually reducing the TCB of our implementation, without changes in efficiency.

	MM4	MM8	MM16	MM32	MM64
PRAM LPZK (Zarith)	10.4	26.4	272.2	7,904.4	467,055.8
PRAM LPZK (Jasmin)	10.7	24	235.2	7,559.4	454,945.7
#gates	192 (80)	1280 (576)	9216 (4352)	69632 (33792)	540672 (266240)

Table 7: Performance analysis of our parallel verified LPZK implementation using Zarith, comparing to the performance obtained by our parallel verified implementation of the same protocol using a Jasmin field arithmetic backend.