Zilch: A Framework for Deploying Transparent Zero-Knowledge Proofs

Dimitris Mouris and Nektarios Georgios Tsoutsos

Abstract

As cloud computing becomes more popular, research has focused on usable solutions to the problem of verifiable computation (VC), where a computationally weak device (Verifier) outsources a program execution to a powerful server (Prover) and receives guarantees that the execution was performed faithfully. A Prover can further demonstrate knowledge of a secret input that causes the Verifier's program to satisfy certain assertions, without ever revealing which input was used. State-of-the-art *Zero-Knowledge Proofs of Knowledge* (ZKPK) methods encode a computation using arithmetic circuits and preserve the privacy of Prover's inputs while attesting the integrity of program execution. Nevertheless, developing, debugging and optimizing programs as circuits remains a daunting task, as most users are unfamiliar with this programming paradigm.

In this work we present Zilch, a framework that accelerates and simplifies the deployment of VC and ZKPK for any application *transparently*, i.e., without the need of trusted setup. Zilch uses traditional instruction sequences rather than static arithmetic circuits that would need to be regenerated for each different computation. Towards that end we have implemented zMIPS: a MIPS-like processor model that allows verifying each instruction independently and compose a proof for the execution of the target application. To foster usability, Zilch incorporates a novel cross-compiler from an object-oriented Java-like language tailored to ZKPK and optimized our zMIPS model, as well as a powerful API that enables integration of ZKPK within existing C/C++ programs. In our experiments, we demonstrate the flexibility of Zilch using two real-life applications, and evaluate Prover and Verifier performance on a variety of benchmarks.

I. Introduction

Cloud computing offers on-demand computational power, emerging as an ideal solution for outsourcing computation from relatively weak devices (phones, laptops, IoT). However, delegating computation to

D. Mouris and N. G. Tsoutsos are with the Department of Electrical and Computer Engineering, University of Delaware, Newark, DE. E-mail: {jimouris, tsoutsos}@udel.edu

an untrusted third-party running unreliable software, and potentially untested – or malicious [1], [2] – hardware, comes with many risks [3]. Data may even be corrupted while at rest [4], or in transit. Moreover, cloud service providers may have monetary incentives to either skip computation steps (e.g., skip computing all decimal points on a big number) or completely counterfeit a result. *How can we trust the results computed by the cloud and be assured that the computation was carried out faithfully?*

Verifiable Computation (VC) leverages mathematical and cryptographic primitives, such as probabilistically checkable proofs (PCPs) [5], [6], interactive proofs [7]–[11] and commitment-based argument schemes [12]–[17], to provide strong guarantees to a client on the correct evaluation of a statement in NP. In these schemes, one party $\mathcal{P}$ (the prover) generates and commits to a proof that the computation was executed faithfully and another party $\mathcal{V}$ (the verifier) performs unpredictable tests to efficiently check the integrity of the execution. Given an honest $\mathcal{P}$, these tests can convince $\mathcal{V}$. Conversely, a faulty execution would be noticed by the verifier with very high probability. Intuitively, in such protocols the overhead for the prover and the complexity of performing the tests for $\mathcal{V}$ should be less than the whole execution on the computationally-weak device; i.e., the verifiable outsourcing is practical. Numerous systems [18]–[20] strive to bring verifiable outsourcing one step closer to practicality.

A notable extension to verifiable computation is to enable the prover to apply the computation on a set of secret inputs – also called *witness* – which are never revealed to $\mathcal{V}$. This approach, known as a *zero-knowledge proof* (ZKP), allows the prover to convince the verifier that she knows a secret without actually disclosing it [21]–[24]. For instance, ZKPs can be leveraged to log-in to a website without typing a password by simply sending a proof that you "know the valid password". ZKP-based authentication eliminates the need for maintaining server-side password databases, sending passwords via unsafe channels, having to digitally sign challenge messages that could later be misused, or even having to disclose an intellectual property for functional verification [25].

Although ZKPs and VC have numerous applications, many state-of-the-art solutions (e.g., [19], [21]–[23], [26]) suffer from a severe limitation: they require a trusted authority to generate the public parameters for the system and then eliminate any knowledge of the randomness used to generate them (referred to as *toxic waste*). A malicious third-party that obtains access to that toxic waste can forge false proofs and trick an honest verifier. Having a single point of failure in VC and ZKP systems that rely on cryptographic primitives seems contradictory. As a result, systems that use public randomness and thus have a *transparent setup* have been proposed [27]–[33]. Likewise, the works in [34], [35] provide constructions leveraging updatable common reference strings (CRS).

Another observation about the computational model followed by most VC and ZKP systems is the need to express computer programs as arithmetic circuits, or equivalently as a set of arithmetic constraints over

a finite field $\mathbb{F}$. Works such as [18], [21], [24] provide a compiler from a high-level language (typically a subset of C) to arithmetic circuits, however, they require the circuit to be fixed offline during the trusted setup phase. Notably, this conversion is laborious as it is hard to express arbitrary algorithms using arithmetic circuits and even harder to edit, debug, or optimize these VC circuits without deep knowledge of cryptography and circuit design. Furthermore, these circuits are program-specific and cannot be reused to verify other programs, which renders them non-universal. An alternative is to employ a Random Access Machine model that provides a low-level language, such as TinyRAM [23], that can define a universal circuit. However, program development using such esoteric machine models without high-level toolchain support still requires significant effort from a programmer's perspective.

Unfortunately, neither of these models of computation is natural for most non-crypto savvy programmers. Thus, an important objective in this work is to develop a methodology for verifiable computation and zero-knowledge proofs of knowledge using a convenient programming model that does not rely on any trusted third party. Our approach is to leverage a programming model that is based on a *sequence of instructions* instead of a circuit netlist. At the same, an additional goal is to optimize performance while ensuring programming convenience.

The main contribution of this work is the development of Zilch[1], a specialized framework to facilitate the development of interactive zero-knowledge proofs for any application. Zilch enables the development of algorithms used for VC and ZKPs using our high-level language called ZeroJava, which is compiled into an intermediate representation (IR) that is ultimately transformed into a set of mathematical constraints. ZeroJava is an intricately chosen subset of Java specifically tailored for deploying zero-knowledge arguments, while the IR statements are evaluated in our custom abstract machine (called ZMIPS) that is adopted from a MIPS processor. To generate and verify proofs, Zilch leverages the state of the art zk-STARK library [29] that does not rely on any trusted third party setup (i.e., contrary to other libraries [22], [23], [26]). Moreover, zk-STARK is resilient against attacks by large-scale quantum computers[2] and its security relies on collision-resistant hash functions [13] and the random oracle model [36]. In addition to newly developed applications that can be developed in ZeroJava, our Zilch framework offers a powerful API that is compatible with C/C++ programs to facilitate embedding VC and ZKPs into existing code. In all cases, Zilch enables a prover to interact with a verifier automatically over a network.

Our contributions are summarized as follows:

[1]**Zilch** / zɪltʃ / : zero; nothing. *The search came up with zilch.*

[2]We refer to a system's property of being resilient to known attacks by large-scale quantum computers as *plausible post-quantum secure*.

- Design of ZMIPS, an abstract machine that is adopted from the MIPS processor with judiciously selected instructions that create arithmetic circuits and enable zero-knowledge proofs for any target application,
- Design and implementation of the ZeroJava high-level language, with a cross-compiler from ZeroJava to ZMIPS and an assembly optimizer,
- Development of the Zilch framework and API to facilitate the construction of ZKPs for existing C/C++ applications.

Roadmap: The rest of the paper is organized as follows: In Section II we discuss background notions, while in Section III we elaborate on our ZeroJava language and cross-compiler, as well as the ZMIPS instruction set architecture (ISA) and the C/C++ API. We demonstrate our framework's capabilities using two real-world applications and evaluate its performance on a variety of benchmarks in Sections IV and V. In Section VI we discuss related works, while our concluding remarks are summarized in Section VII.

II. Preliminaries

A. Models of Computation

There exist many different models of computation, some less powerful yet simpler, while others are more sophisticated. In the context of this article, we delve into two models that enable the execution of arbitrary computer programs: Turing machines (TMs) and arithmetic circuits (ACs).

A Turing Machine is a model of computation that consists of an infinite tape, a tape head, and a finite table of rules. At each step, the tape head reads a symbol from the tape and determines which action to perform from the finite table and then either moves one cell to the left or right or halts the computation. This abstract machine, despite its simplicity, is capable of executing any algorithm given as a set of rules for an input provided in the tape [37]. A universal Turing machine (UTM) is a TM whose algorithm (table of rules) implements *a simulator* for any arbitrary TM with arbitrary input tape. A fundamental difference between a TM and a UTM is that the former is programmed with a rules table to evaluate a specific problem, while the latter works with the description of any TM and thus can evaluate any program.

An Arithmetic Circuit over a field $\mathbb{F}$ consists of input and output gates that are connected with intermediate gates through wires. The input values proceed through a sequence of gates performing either addition ($+$) or multiplication ($\times$); a simple example is illustrated in Fig. 1. Transforming certain classes of programs into ACs can be straightforward if they only involve addition and multiplication of elements of the finite field. Notably, this approach is equivalent to the evaluation of polynomials over a

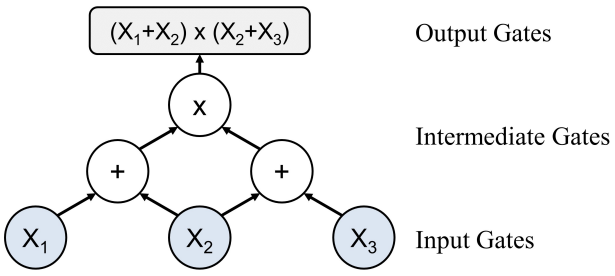


Fig. 1: Example of a computation using arithmetic circuits. The output can be expressed using a polynomial expression.

field $\mathbb{F}$, so the outputs of the arithmetic circuit can be expressed as a set of polynomials over the input variables.

Turing machines and ACs are equivalent models of computation, i.e., given a program and an input, both models can compute the same output. In fact, any Turing machine can be unrolled into a circuit somewhat larger than the number of steps in the computation [38]. Our abstract machine in Zilch offers a more flexible model of computation since it is the equivalent of a UTM: its input is a program defined as a sequence of instructions that can consume any given input.

B. Principles of ZKPs and VC

Verifiable Computation: The typical scenario in VC is that the verifier ($\mathcal{V}$) sends a program description Ψ and an input $\mathbb{x}$ for that program to the prover ($\mathcal{P}$). Then, $\mathcal{P}$ computes and returns the output $\mathbb{y} = \Psi(\mathbb{x})$ of the execution of that program on input $\mathbb{x}$ to $\mathcal{V}$ along with a short proof which can be efficiently verified by $\mathcal{V}$. In this case, both parties express the computation Ψ as a set of constraints involving $\mathbb{x}$ and $\mathbb{y}$. Those constraints are essentially equations over a finite field $\mathbb{F}$ modulo a large prime. Consecutively, $\mathcal{P}$ solves the constraints (i.e., finds a satisfying assignment), where a solution exists if and only if $\mathbb{y} = \Psi(\mathbb{x})$. These constraints are equivalent to ACs [21], [23], where the gates are operations in $\mathbb{F}$ and the wires are elements in $\mathbb{F}$.

Zero-Knowledge Proofs: To also make the aforementioned short proofs privacy-preserving, $\mathcal{P}$ can provide her own private input $\mathbb{w}$ to the computation, referred to a *witness*. Thus, Ψ now becomes a function of two inputs such as $\mathbb{y} = \Psi(\mathbb{x}, \mathbb{w})$. If $\mathcal{V}$ can be convinced that the statement $\mathbb{y} = \Psi(\mathbb{x}, \mathbb{w})$ is True without learning anything about $\mathbb{w}$, then the scheme is a ZKP protocol; such protocols become more powerful when the witness is a solution to an NP-hard problem. Most existing works leverage ACs where the algorithm is transformed to constraints and the proof convinces $\mathcal{V}$ that there exists a witness

satisfying these constraints. Nevertheless, an important limitation of earlier works (e.g., [18]) is the need to know the target NP-hard algorithm beforehand, rendering them non-universal.

C. Properties of Proof Systems

Every proof system should satisfy two basic properties:

- **Completeness:** If the statement $\mathbb{y} = \Psi(\mathbb{x})$ is True, an honest $\mathcal{P}$ should be able to convince an honest $\mathcal{V}$. In other words, given the same set of inputs, $\mathcal{V}$ should yield the same result $\mathbb{y}$ through the protocol as $\mathcal{P}$.
- **Soundness:** If the statement $\mathbb{y} = \Psi(\mathbb{x})$ is False, a malicious $\mathcal{P}$ cannot convince an honest $\mathcal{V}$ that it is True (except with negligible probability).

If the proof system also preserves the privacy of the prover's inputs, then it would satisfy the zero-knowledge property:

- **Privacy:** If the statement $\mathbb{y} = \Psi(\mathbb{x}, \mathbb{w})$ is true, $\mathcal{P}$ can convince $\mathcal{V}$ without leaking any information about $\mathbb{w}$.

In ZKP systems, we have two additional desired properties:

- **Transparency:** The proof system does not require any trusted setup (e.g., [28]–[30], [32]); any randomness used by transparent frameworks is public coins.
- **Scalability:** The proof system can gracefully handle programs and inputs of larger sizes, which makes it more practical. This property is applicable to both the prover and the verifier: The scalability of $\mathcal{P}$ corresponds to the overhead of generating the proof and convincing $\mathcal{V}$, and it should be somewhat similar to the time it would take to re-execute the target program. Likewise, the scalability of $\mathcal{V}$ entails that verification times are exponentially smaller than the cost of re-executing the target program (scalable verifiers are referred to as *succinct*).

A proof system that satisfies all the above properties is a *Zero-Knowledge Scalable Transparent ARgument of Knowledge (zk-STARK)* [28], [29].

D. A Primer on zk-STARKs

Overview: Typically, a ZKPK involves (1) a *witness* input $\mathbb{w}$ (i.e. the piece of data we want to prove knowledge for), and (2) verifiable execution of a public algorithm that tests an assertion about $\mathbb{w}$. For example, the latter can be an algorithm $\mathbb{A}$ that "multiplies two integers and compares the result with a composite N," whereas the witness can be a set of primes p, q. In ZKPK, we can prove knowledge of correct p and q if we can prove that $\mathbb{A}$ was executed faithfully on the witness input. Likewise, $\mathbb{A}$ can be a *modular Fibonacci loop* instantiated with integers a_0, a_1 so that each loop computes $a_i =$

$a_{i-1} + a_{i-2} \bmod p$, where p is a large prime that defines a finite field. In this case, if $a_0 = 1$ and $a_1 = \mathbb{w}$ is our witness, we can prove knowledge of a suitable a_1 the returns the anticipated output $\mathbb{y}'$ after a large number of loops (i.e., $a_T = \mathbb{y}'$ at step T).

The zk-STARK methodology enables proving the integrity of the computation of algorithm $\mathbb{A}$ after T steps on input $\mathbb{w}$ that yields an output $\mathbb{y}$; this is possible using *arithmetization* and *low-degree testing* operations on polynomials over finite fields [29]. Specifically, arithmetization is the reduction of a computational problem (i.e., verifying a computation) into an algebraic problem, such as checking that a certain polynomial is of low degree. In zk-STARKs, arithmetization comprises three steps: (a) generating the execution trace of algorithm $\mathbb{A}$ for T steps, (b) generating a set of polynomials that express constraints for each execution step (e.g., if two values are multiplied, the result equals their product), and (c) combining the execution trace and polynomial constraints into a single polynomial $\mathbb{Q}$. Finally, the zk-STARK approach shows that it is possible to generate a ZKPK by employing error-correction methods (specifically Reed-Solomon proximity testing [29]) and show that the generated polynomial $\mathbb{Q}$ is actually low-degree. In this case, $\mathcal{V}$ is convinced that a polynomial is of low-degree (and thus the integrity of the computation of $\mathbb{A}$) with only a small number of queries to $\mathcal{P}$ [29].

Low-Degree Extension and Commitment: In the first step in a zk-STARK proof, $\mathcal{P}$ encodes the execution trace of algorithm $\mathbb{A}$ into a sequence of states. In our earlier modular Fibonacci loop example, this would be the set $\{a_0, a_1, a_2, \ldots, a_T\}$ (note, $\mathcal{P}$ knows the correct witness a_1). $\mathcal{P}$ then generates a sequence $\{b_0, b_1, \ldots\}$ and pairs each state in the trace with the corresponding b_i (e.g., create (b_i, a_i) pairs). These pairs are interpreted as (x, y) points and are used to efficiently compute the *interpolating trace polynomial* $F(B)$ across them with the Lagrange interpolation method. Knowing the coefficients of the trace polynomial, $\mathcal{P}$ fixes an R so that $R \gg T$ computes its *low-degree extension (LDE)*, i.e., the values of $F(B)$ for $B = \{b_{T+1}, b_{T+2}, \ldots, b_R\}$. Finally, $\mathcal{P}$ computes a Merkle-tree over the sequence $\{F(b_0), F(b_1), \ldots, F(b_R)\}$ and commits to the tree root.

Arithmetization: zk-STARK employs a formal algebraic intermediate representation (AIR) of the target algorithm $\mathbb{A}$ as a set of low-degree polynomials $\{P_1, \ldots, P_K\}$ that encode K constraints about the execution trace of the computation [29]. In this case, the transition from step T to step $T+1$ in the computation is valid if and only if all constraints are satisfied (i.e., $P_1 = \cdots = P_K = 0$). In our Fibonacci example (with interpolated trace polynomial F), our constraints are $F(x+2) - F(x+1) - F(x) = 0$ for $x \in \{b_0, b_1, \ldots, b_{T-2}\}$, $F(x) - 1 = 0$ for $x = b_0$, and $F(x) - \mathbb{y}' = 0$ for $x = b_T$, where all operations are $\bmod\ p$.

By the *polynomial remainder theorem*, if b is a root of a polynomial $Q(x)$, then $Q(x) = (x-b)P(x)$, i.e., $P(x) = Q(x)/(x-b)$ is a polynomial. Therefore, we can can compute the AIR polynomials by

factorizing the constraints of the computation. In our Fibonacci example we have $P_1(x) = (F(x) - 1)/(x - b_0)$, $P_2(x) = (F(x) - 1)/(x - b_T)$, and $P_3(x) = (F(x+2) - F(x+1) - F(x))/[\prod_{i=0}^{T-2}(x - b_i)]$. Moreover, zk-STARK computes the *Composition Polynomial (CP)* as the linear combination of all P_is and applies a LDE up to R points.

Low-degree testing: The goal of this step is to convince $\mathcal{V}$ that the *distance* between the computed CP and a low degree polynomial is relatively small, where the distance between any function and a polynomial of degree d is defined as the number of x inputs where their values are different. In zk-STARK, this is possible using the *FRI operation* (Fast Reed-Solomon Interactive Oracle Proofs of Proximity), which reduces the problem of proving that a function of domain size R is close to polynomial of degree bounded by d into a new smaller problem where the function domain size is $R/2$ and the polynomial degree is $d/2$ [29]. FRI is applied iteratively to CP (treated as a function of domain size R after LDE) until $d = 1$; each iteration replaces every polynomial power x^i with $x^{\lfloor i/2 \rfloor}$ and the coefficients of same powers are added. Also, after each FRI iteration, $\mathcal{P}$ computes a Merkle-tree of the values of CP over its entire domain and the tree root is committed.

Decommitment Step: In this step, $\mathcal{V}$ is convinced that the original execution trace for algorithm $\mathbb{A}$ was computed faithfully by sending queries to $\mathcal{P}$. Specifically, the verifier selects a small set of b_i values for $i \in \{0, 1, \ldots, R\}$ and for each one the prover reveals the Merkle-tree path for her commitments for each FRI step and the LDE of the trace polynomial F. $\mathcal{V}$ uses the values of F and reconstructs the corresponding CP values (for each FRI step). If all commitments are correct, $\mathcal{V}$ is convinced the proof is sound with very high probability [29].

E. Our Threat Model

Cheating Prover: To mitigate the risks applicable to our approach for computational integrity, our threat model assumes an adversary given access to the prover's capabilities. The adversary succeeds if she produces a false statement that will convince $\mathcal{V}$ to accept it. In the VC scenario, the cheating $\mathcal{P}$ has incentives to skip some steps or completely forge the result, while in the ZKPK case the adversary tries to convince $\mathcal{V}$ that she knows the witness without actually knowing it. Zilch features a configurable security parameter λ, which determines the probability ($\leq 2^{-\lambda}$) that an adversary can successfully deceive an honest verifier in the above experiments. Thus, λ defines the soundness property of the proof system.

Cheating Verifier: On the other hand, a *malicious* verifier is assumed to behave without restrictions and not necessarily follow the protocol specification in order to extract any information about the secret input $\mathbb{w}$. If $\mathcal{P}$ follows the protocol correctly and the statement is true, $\mathcal{V}$ will never learn any (private) witness data from the interaction with the prover except the fact that the statement is true, (i.e., zero-knowledge

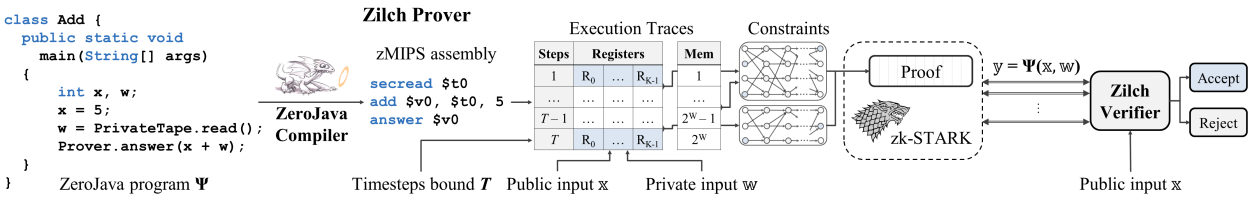


Fig. 2: **Zilch Framework Overview:** Using our ZeroJava compiler, $\mathcal{P}$ provides the assembly code to Zilch along with the private and public inputs. Zilch first determines the steps bound T automatically and then computes the result $\mathbb{y}$. Finally, $\mathcal{P}$ and $\mathcal{V}$ interact over a limited number of rounds and the verifier either accepts the proof (i.e., she is convinced) or rejects it.

property). Moreover, we don't consider trivial cases where $\mathcal{V}$ completely withdraws from the protocol of the two parties.

Post-quantum resilience: Many different VC and ZKP frameworks rely on elliptic curves and pairings [15], [21]–[23], [26], [39]–[42], as well as the discrete logarithm problem [30], [43]; this makes them susceptible to attacks using quantum computers [44]–[47]. Aurora [28], Ligero [27], and zk-STARKs [29] rely on collision-resistant hash functions, which renders them resilient to known attacks from quantum computers. Likewise, since Zilch utilizes zk-STARK as its back-end proof system, it inherits the same post-quantum resilience property.

Transparency: Previous VC and ZKP solutions (e.g., [18], [21]–[23], [26], [41]) require a third party to set-up the system with non-public randomness. If that party is not trustworthy, this secret randomness could be misused to generate false proofs and compromise the system's security. Zilch, like other state-of-the-art PCP-based systems (e.g., [27], [29], [32]) is transparent as it relies *only on public randomness* and does not need any trusted third party during its set-up phase.

III. The Zilch Framework

A. Key Observations in our Methodology

Zilch aims to facilitate proving computational integrity statements; in particular, our goal is to convince a verifier that a computer program implemented as a sequence of well-defined instructions returns an expected output for a set of inputs. For our methodology, we observe that in order to verify the execution integrity of an algorithm, it is sufficient to *divide it into two parts, verify both individually and finally verify a correct transition from the first part to the second.* This observation can be applied in a divide-and-conquer manner to recursively decompose any algorithm into sub-algorithms until each becomes simple enough to be verified individually. Each individual proof is then combined into a composable proof for the execution of the original algorithm.

A second important observation about computational integrity in our case is that it is sufficient to decompose the target algorithm up to *the granularity of individual assembly instructions* and prove the integrity of each instruction directly using its corresponding arithmetic circuit (AC). This offers great flexibility, as a predetermined set of assembly instructions (each mapped to a small AC) can be combined to define any arbitrary algorithm. Conversely, trying to verify any large program directly using a large static AC would require generating a unique AC for each different program, which is exactly the daunting task that we are trying to avoid in the first place. Notably, our proposed method of verifying programs at the granularity of an assembly instruction is beneficial, as it is relatively easy to translate any program written in a high-level language into a set of assembly instructions using a compiler. To prove the integrity of execution, we first verify the AC of each individual assembly instruction and finally verify each state transition between consecutive instructions.

B. Overview of our Framework

To instantiate our methodology, we have developed Zilch: a transparent and post-quantum resilient programming framework for creating ZKPK for any application. Zilch is universal since it takes as input *a description of a Turing Machine (TM)* (i.e., a computer program) and *two input tapes*, one private and one public. More formally, Zilch implements a time-bounded Universal TM and can be used for any arbitrary computation that is expressed as a sequence of instructions. Internally, Zilch adopts a MIPS-like processor model (i.e., an abstract machine with memory, program counter, registers, and fetch-decode-execution pipeline stages) called ZMIPS; our machine supports a judiciously selected instruction set that can implement and verify any computation in zero-knowledge.

Zilch consists of a front-end and a back-end. The front-end defines our customized subset of Java specifically tailored to zero-knowledge arguments, called ZeroJava, and includes our compiler for translating the ZeroJava high-level code into ZMIPS assembly instructions. From a programmer's perspective, ZeroJava is *object-oriented and strongly-typed* like Java, while excluding Java features that complicate the run-time system, such as exceptions and multi-threading. Our compiler comprises four phases: (a) transforming the high-level code into an intermediate representation (IR), (b) performing static analysis on the IR to optimize it, (c) performing register allocation to minimize the number of required registers, and (d) generating the ZMIPS assembly. We elaborate more on the design choices of the Zilch front-end (i.e., the ZeroJava language and the compiler) in Section III-C.

The Zilch back-end defines the ZMIPS abstract machine that consumes ZMIPS instructions to transition from one state to another; *each state comprises the program counter, K registers, and memory.* A computation is expressed as a sequence of instructions or equivalently as a sequence of abstract machine

states. Furthermore, each assembly instruction generates individual constraints that must hold between each two consecutive abstract machine states, and having a finite set of instructions renders verification feasible. The sequence of states $\{S_1, S_2, ..., S_T\}$ forms an execution trace of a program (also called a *transcript*) that is T steps long; in each step, an instruction is fetched, decoded and executed by our abstract machine. The transcript can be represented as a two-dimensional table with T rows and K columns (Fig. 2), where each row represents a single execution step and each column tracks one ZMIPS register through time. Two states S_i and S_{i+1} are valid if the machine in state S_i can transition with some instruction to state S_{i+1} in the next step. Depending on the instruction that operates on S_i, the new S_{i+1} state is different, since each instruction performs a unique transition from S_i to S_{i+1}. Given a time bound T, an execution trace tr of a specific program Ψ is valid if there exist public and private inputs $\mathbb{x}$, $\mathbb{w}$, such that the generated trace of Ψ on inputs $\mathbb{x}$ and $\mathbb{w}$ is tr. Likewise, the integrity of the memory state is ensured using a memory trace as will be discussed in Section III-D. For each ZMIPS instruction, our Zilch back-end invokes the corresponding AIR constraints employing the zk-STARK library [29] (described in Section II-D); using this library, Zilch consumes the transcript and the constraints, generates a low-degree polynomial, and then $\mathcal{P}$ is able to convince $\mathcal{V}$ that all polynomial constraints are satisfied in the execution trace for a secret witness $\mathbb{w}$.

Benefits of Zilch: Expressing a computation as a transcript of state transitions enables our abstract machine to generate universal ACs that do not require a different set up each time a new program is executed since each instruction implements its own AC. The only requirement is to provide an upper bound for the time steps of the target program in order to generate an AC that simulates the entire execution. Zilch can automatically determine the minimum number of execution time steps required for a specific program Ψ to generate a result by first simulating the computation quickly (without generating any constraints or proofs), and then checking if the output matches the expected result $\mathbb{y}$. If not, Zilch doubles the time steps bound T and repeats the same check with the new bound. If the computation returns $\mathbb{y}$ using the new time steps bound, Zilch generates the AIR constraints for the identified bound T and interacts with the verifier.

The zk-STARK library enables the development of interactive proofs where the prover and the verifier communicate over several rounds until the latter is convinced of the correctness of a proof. As we illustrate in Fig. 3, $\mathcal{P}$ and $\mathcal{V}$ initially agree on the polynomial constraints for the computation and then the prover generates the transcript and sends its encoding to the verifier. As soon as the verifier confirms the consistency of the encoding, the two parties interact over several rounds; in each round $\mathcal{V}$ first sends a message to $\mathcal{P}$ comprising *public random coin-flips*, and then $\mathcal{P}$ replies with *an oracle* (i.e., a long message comprising a proof) that the verifier can query probabilistically at any index of her choice.

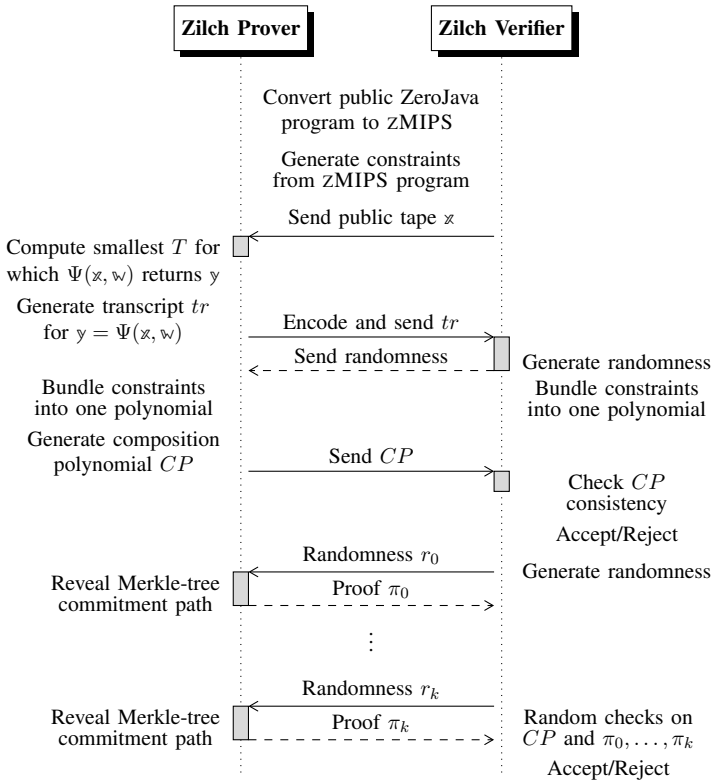


Fig. 3: **Prover and Verifier Interaction.** Starting from a public computation expressed in ZeroJava and a public tape, $\mathcal{P}$ and $\mathcal{V}$ agree on the polynomial constraints. Next, $\mathcal{P}$ generates and encodes the transcript tr, and combines it with the polynomial constraints into a single composition polynomial CP that is shared with $\mathcal{V}$. Finally, the two parties engage in the FRI protocol and $\mathcal{V}$ either accepts or rejects $\mathcal{P}$'s statement.

As Zilch employs the zk-STARK protocol for verification, we realize this oracle using Merkle tree commitments, and the corresponding cost for $\mathcal{V}$ is poly-logarithmic in the time steps required to execute Ψ, which renders our framework *succinct*.[3] Another important benefit of Zilch is its ability to serialize all the rounds of interaction between $\mathcal{P}$ and $\mathcal{V}$ that are shown in Fig. 3, which enables zero-knowledge proof verification over a network. In Section V, we present how the communication cost scales with an increasing number of zMIPS instructions.

An overview of the Zilch framework is presented in Fig. 2, where a simple ZeroJava program for addition is translated into zMIPS assembly instructions using our compiler. Zilch then produces an execution trace that is ultimately transformed into AIR constraints for the zk-STARK library, and $\mathcal{P}$ can interact with $\mathcal{V}$. For an honest prover, the verifier is convinced (i.e., accepts the proof) that the program

[3] *Succinctness* denotes short proofs and scalable verification time.

was executed faithfully and that $\mathtt{y} = \Psi(\mathtt{x}, \mathtt{w})$ after at most T steps. Conversely, if the prover is malicious, the verifier will reject the proof with very high probability.

C. Zilch Front-End Design

1) ZeroJava Language: To facilitate the development of ZKPK for any application, we introduce a self-contained high-level language called *ZeroJava*, which enables implementing arbitrary NP statements and is specifically tailored to VC and ZKPK. Contrary to previous approaches on high-level languages for verifiable computation (e.g., [18], [21], [24]), ZeroJava supports dynamic loop conditions without the need for unrolling (i.e., mutable state and iteration, dynamic termination, and infinite loops are supported). Moreover, ZeroJava supports ZKP-specific built-in functions that invoke specific ZMIPS assembly instructions (we elaborate on these methods in Section III-D that discusses the ZMIPS ISA). The following paragraphs present our design choices for ZeroJava.

ZeroJava is object-oriented and strongly-typed, like Java. The basic types of ZeroJava are `int` for W-bit signed integers, `boolean` for logical values, and `int[]` for arrays of integers. Integers are represented using two's complement, and overflows wrap around as in standard Java. Classes contain attributes and methods with arguments and return type of basic or class types. ZeroJava supports single inheritance without interfaces and function overloading (i.e., each method name must be unique). In addition, all methods are inherently polymorphic so that a method can be defined in a subclass if it has the same return type and arguments as in the parent. Fields in the base and derived class are allowed to have the same names and are essentially different fields. All ZeroJava methods are public and all fields are protected so that a class method cannot access fields of another class, with the exception of its parent; a class's own methods can be called via `this`. Local variables can be defined at the beginning of a method and can shadow the fields of the surrounding class with the same name.

In ZeroJava, the `new` operator calls a default void constructor. In addition, there are no inner classes and there are no static methods or fields. A ZeroJava program begins with a special main class that does not have fields and methods and contains the main method (i.e., `public static void main(String[] args)`). After the main class, other classes may be defined that can have fields and methods. In Table I we summarize all the ZeroJava supported operators.

Tapes: ZeroJava supports both public and private inputs via two read-only input files called *tapes*. Each tape can be read sequentially using the `READ` built-in method (the next word is consumed), or with the random access `SEEK` function (the word at a given offset is read). These built-in methods have one-to-one correspondence with ZMIPS instructions. In the case of ZKPK, the secret input (witness `w`) should be provided in the private tape; for VC, only the public tape is required.

TABLE I: ZeroJava Language Operators

Assignment	Increment & Decrement	Arithmetic & Bitwise	Logical & Relational
`a = b`	`a++`	`a + b`	`!a`
`a += b`	`a--`	`a - b`	`a && b`
`a -= b`	**Arrays**	`a * b`	`a || b`
`a *= b`	`a[b]`	`a / b`	`a == b`
`a /= b`	`a.length`	`a % b`	`a != b`
`a %= b`	`new int[a]`	`a ^ b`	`a < b`
`a ^= b`	**Ternary**	`a & b`	`a > b`
`a &= b`	`(a) ? b : c`	`a | b`	`a <= b`
`a |= b`		`~a`	`a >= b`
`a <<= b`		`a << b`	
`a >>= b`		`a >> b`	

ZeroJava Example: In Fig. 4, we provide a ZeroJava program that implements Wegner's efficient algorithm [48] to compute the Hamming weight of a secret number and then compare it with a public threshold. This example highlights various features of ZeroJava, such as reading from the public and the private tape, performing loops, as well as applying arithmetic, logical, and bitwise operations. Besides, a potential application of this algorithm could be to compute the Hamming weight of a private RSA exponent in zero-knowledge and convince a verifier that it is greater than a threshold; this could offer additional assurance against certain attacks (e.g., the authors of [49] demonstrate a birthday attack on RSA private exponents with low Hamming weight).

2) ZeroJava Compiler: Using our ZeroJava compiler, programmers can translate NP statements expressed in ZeroJava high-level code into optimized ZMIPS machine code. Since ZeroJava is a strongly-typed language, the first step performed by our compiler is to statically analyze the program and verify its type safety, i.e., ensure that the types of expressions are consistent. For instance, a variable declared as an integer cannot be assigned with a different data or class type on the same scope. To detect syntax errors, our compiler performs multiple visits on the ZeroJava code to first extract the classes information, then generate a symbol table, and finally check the static type-safety of all the expressions in the high-level code. Our compiler also throws an error if an `answer` function is missing, as this is required to halt the abstract machine. Consecutively, the ZeroJava compiler parses the high-level code, generates an IR that is in turn consumed by the code optimizer. In particular, our code optimizer reduces the IR code based on the results of static analysis, employing data-flow analyses and optimization techniques including live-

```
class HammingWeightThreshold {
  public static void main(String[] a) {
    int threshold = PublicTape.read();
    int num = PrivateTape.read();
    int count = 0;
    while (num > 0) {
      num &= num - 1;
      count++;
    }
    Prover.answer(count > threshold);
  }
}
```

Fig. 4: ZeroJava program to prove that a secret number has a Hamming weight that is greater than a public threshold.

range, dead-code, constant- and copy-propagation [50]. Finally, our compiler performs register allocation on the IR to further reduce the number of registers and generates zMIPS assembly. Our optimizations based on IR static-analysis can be summarized as follows:

- **Live range analysis:** The *liveness* analysis determines which variables hold a value that may be needed in the future (i.e., are live) for each instruction. This is used for the dead code elimination optimization (discussed next).
- **Dead code analysis:** An assignment to a non-live variable is *dead code*; such assignments can be removed, reducing the total size of the program.
- **Constant propagation:** For each program instruction, this analysis determines which variables hold a *constant value*. In this case, the constant value is forwarded to all subsequent uses of the variable.
- **Copy propagation:** Likewise, this analysis determines which program variables are guaranteed to hold identical values. Both constant and copy propagation analyses enable further dead code elimination optimizations.

These optimizations are executed until a fixed-point is reached (i.e., a steady state where two consecutive iterations result in the same code sequence); then, no further optimizations can be detected by static analyses. In this work, we employ the Datalog declarative logic programming language from within the IRIS framework [51]. Since Datalog naturally supports recursive relations, it is suitable for fixed-point algorithms [52]. In our case, after the ZeroJava compiler has generated the IR, our code optimizer parses the code and generates relation tables (e.g., simple-instruction, jump-instruction, next-instruction, etc.)

that are used for static analysis in Datalog.

Naturally, the object-oriented paradigm comes with a performance trade-off when it is applied to zero-knowledge statements since instantiating new objects requires creating virtual tables and accessing the memory. Therefore, our ZeroJava compiler minimizes any unnecessary memory operations when objects are not used and the statements are only in the main class. Furthermore, the combination of the static analysis optimizations and register allocation techniques of the ZeroJava compiler are crucial since they minimize the number of registers that are *spilled* (i.e., having to move their values to and from memory). Moreover, as the number of instructions affects the time steps bound, minimizing the total number of ZMIPS instructions results in faster proving time.

Debugging: ZeroJava also features a `System.exit(int)` method that can be used to terminate the execution of a ZeroJava program and at the same time return a status code for debugging purposes. Notably, the `System.exit(int)` method is intended to be used solely for debugging and does not replace the `answer` method. Additionally, to enable the advanced debugging techniques of Java, such as breakpoints and the Java debugger (`jdb`), Zilch provides a preprocessor that automatically transforms any ZeroJava program to pure Java code by converting any Zilch-specific statements to the equivalent ones in Java. Specifically, our debugging preprocessor converts the `answer` function to a `System.out.println` invocation followed by a `return` statement, whereas the methods that read from the tapes are replaced with standard Java methods that read from files.

D. Zilch Back-End Description

Instruction Execution: Each instruction can modify one or more registers, the program counter, and the memory, populating a new row in the transcript, while its corresponding AC defines constraints and assertions for these transitions (both the prover and the verifier agree on these in advance). To ensure correct instruction execution (i.e., *code-consistency*), for each step i the transition between consecutive machine states (S_i, S_{i+1}) is verified by the AC corresponding to instruction i based on the following assertion: Executing instruction i on state S_i results in a new S_{i+1} state, where the destination register in instruction i, as well as the program counter, are updated according to the instruction operation code and all the values on the other registers are propagated to the next state. Each instruction increments the program counter by one after it is executed, except for `jump` instructions that modify the program counter based on the branch target.

Considering pairs of adjacent states in a time-sorted transcript, code consistency can be checked by inspecting one pair at a time. For instance, after a `move dst, src` instruction is executed, the value in the destination register (`dst`) should be equal to the value of source register (`src`) before executing the

instruction, and all other registers should remain the same. Such constraints should be satisfied between two consecutive states at the execution trace for each `move` instruction. In a `jump` instruction, the consistency of the program counter is asserted while all other registers should remain the same. In a similar manner, we handle the constraints for all instructions that do not involve memory. Initially, the program counter (PC) is set to 0 and the first instruction is fetched; subsequently, each instruction i that is fetched is always pointed by the PC.

Memory Accesses: The back-end of Zilch employs the zk-STARK library to transform the execution trace and the polynomial constraints into a single low-degree polynomial and convince the verifier of their satisfiability over the specific execution trace, which guarantees computational integrity. Similar to code consistency, memory consistency is ensured using constraints on pairs of adjacent states; these states are encoded in a memory transcript sorted by ascending memory locations and then by time. If i is a `load` instruction at a specific address, the value read by i should equal the last value written to that address by the most recent `store` instruction.

By analyzing both the code and memory transcripts, it is possible to verify the consistency of all instructions and memory locations respectively during execution. Specifically, zk-STARK enables $\mathcal{P}$ to convince $\mathcal{V}$ that both transcripts correspond to the same program execution (i.e., they encode the same computation) using a *permutation between the two traces* [23], [53]. This permutation is unknown to $\mathcal{V}$ and is verified by zk-STARK via a back-to-back De Bruijn graph, as discussed in [29]. In general, if a program does not use memory-type instructions, the proof comprises fewer constraints, and its execution overhead can be reduced. Conversely, if the program accesses memory, additional constraints are necessary to verify memory integrity, which can impact performance; in fact, as the time bound T increases, the execution time of a program with memory accesses is dominated by the cost of verifying the aforementioned permutation constraints. If all constraints hold during execution and the program finishes within T steps, $\mathcal{V}$ would accept the proof.

1) zMIPS *Assembly Language:* In this work, our goal is to define an instruction set architecture (ISA) for the abstract machine of Zilch that is specifically tailored to VC and ZKPK. This means that our candidate instruction set should (a) be sufficiently simple so that the arithmetic circuit corresponding to each instruction would be easy to evaluate, and (b) have a reduced number of instructions so that the number of unique ACs is also minimized. Some modern instruction set architectures, however, such as the x86, implement a large number of instructions that define low-level or compounded operations (e.g., load a value from memory, then multiply it by 2 and finally store it back to memory), or even operate at multiple elements at once. Such complex ISAs are not suitable candidates for our abstract machine; instead, our goal is to define a reduced instruction set computer (RISC) architecture that is compatible

TABLE II: zMIPS instructions: R_D denotes the destination register, R_S and R_T denote the source registers, A can be either a source register or an immediate value, while L can be either an instruction number or a label.

Arithmetic Operations		
ADD	R_D, R_S, A	$R_D = R_S + A$
SUB	R_D, R_S, A	$R_D = R_S - A$
MULT	R_D, R_S, A	$R_D = R_S \times A$
DIV	R_D, R_S, A	$R_D = R_S \div A$
MOD	R_D, R_S, A	$R_D = R_S \mod A$
MOVE	R_D, A	$R_D = A$
LA	R_D, L	$R_D = L$
Bitwise Operations		
AND	R_D, R_S, A	$R_D = R_S \,\&\, A$
OR	R_D, R_S, A	$R_D = R_S \mid A$
XOR	R_D, R_S, A	$R_D = R_S \oplus A$
NOT	R_D, R_S, A	$R_D = \sim A$
SLL	R_D, R_S, A	$R_D = R_S << A$
SRL	R_D, R_S, A	$R_D = R_S >> A$
Jumps, Branches and Comparisons		
BEQ	R_S, R_T, L	if $R_S = R_T$ then goto L
BNE	R_S, R_T, L	if $R_S \neq R_T$ then goto L
BLT	R_S, R_T, L	if $R_S < R_T$ then goto L
BLE	R_S, R_T, L	if $R_S \leq R_T$ then goto L
SEQ	R_D, R_S, A	$R_D \leftarrow$True if $R_S = A$
SNE	R_D, R_S, A	$R_D \leftarrow$True if $R_S \neq A$
SLT	R_D, R_S, A	$R_D \leftarrow$True if $R_S < A$
SLE	R_D, R_S, A	$R_D \leftarrow$True if $R_S \leq A$
J	L	goto instruction L
JR	R_S	goto instruction denoted by R_S
Load and Store Operations		
LW	R_D, $A(R_S)$	$R_D = MEM[R_S + A]$
SW	R_S, $A(R_D)$	$MEM[R_D + A] = R_S$
I/O Operations		
PUBREAD	R_D	R_D fetch next word from public tape
SECREAD	R_D	R_D fetch next word from private tape
PUBSEEK	R_D, A	R_D fetch word from public tape$[A]$
SECSEEK	R_D, A	R_D fetch word from private tape$[A]$
PRINT	R_S	print R_S
EXIT	R_S	throw exception and return R_S
ANSWER	R_S	return R_S and halt

with ACs in VC and ZKPK.

For our ZMIPS ISA, a natural candidate would be to adopt the MIPS ISA that is sufficiently simple yet very expressive, open-source, and widely used [54]. Moreover, since data memory accesses entail evaluation of additional constraints (as discussed in the previous paragraphs), our ideal ISA should be *register-to-register* and follow the Harvard paradigm with independent memory spaces for instructions and data. Towards that end, we have developed a MIPS-like ISA that includes support for arithmetic, bitwise, comparison, conditional, memory, and I/O operations. In particular, the ZMIPS architecture extends the traditional MIPS ISA with a set of custom I/O instructions for reading public as well as private (witness) data from the input tapes (both sequentially and with random access), as well as instructions to print results and halt.

TABLE III: ZeroJava Built-in Functions

Built-in function	**zMIPS instruction**
`Prover.answer(int)`	ANSWER R_S
`System.exit(int)`	EXIT R_S
`System.out.println(int)`	PRINT R_S
`int PublicTape.read()`	PUBREAD R_D
`int PrivateTape.read()`	SECREAD R_D
`int PublicTape.seek(int)`	PUBSEEK R_D, A
`int PrivateTape.seek(int)`	SECSEEK R_D, A

Instructions: In Table II we present a subset of the assembly instructions supported by ZMIPS. In our notation, register R_D denotes the destination register, while R_S and R_T denote the source registers. Like in the MIPS architecture, our instructions are divided into three broad categories: *R-type* that involves instructions with up to three registers, *I-type* for instructions involving up to two registers and an immediate value, and *J-type* for instructions involving up to two registers and a jump target. In ZMIPS, we simplified the MIPS ISA by merging the *I* and *R* types, however, we still support the *I-type* instructions (not shown in Table II) for backward compatibility with MIPS programs. Most instructions operate on parameter A, which can be either a source register or an immediate value; in this case, Zilch can distinguish *R-type* from *I-type* automatically. In *J-type* instructions, L denotes either an instruction number or a label (as discussed in Section III-D2). Overall, ZMIPS supports arithmetic (i.e., +, -, *, /, `mod`), bitwise (&, |, ^, $\sim$, <<, >>), logical (!, &&, ||), relational ($=$, $\neq$, $>$, $<$, $\geq$, $\leq$), branch/jump, memory transfer and I/O instructions. Additionally, the 1-to-1 mapping between the ZeroJava built-in methods and ZMIPS instructions is summarized in Table III.

Registers: Inspired by the MIPS ISA, ZMIPS supports general-purpose ({`$s0`, `$s1`, ...} and temporaries {`$t0`, `$t1`, ...}) and special-purpose registers (SPRs) such as: the `$zero` (or `$0`) register that is hardwired to zero, the `$ra` register that holds return addresses, the stack `$sp` and frame `$fp` pointer registers that are used to enable the call stack of our abstract machine, `$a0` – `$a3` that store call arguments, and `$v0` – `$v1` that store return values. We further introduce the heap pointer `$hp` SPR that is used to store the next free memory address; we utilize `$hp` to perform dynamic memory allocation in our abstract machine instead of the MIPS system calls.[4] Since ZMIPS is an abstract machine, we can increase its total number of registers to more than the 32 used in MIPS. Thus, the abstract machine state comprises a W-bit program counter and up to K registers of size W bits (all initialized to zero); both the word size W and the total number of registers K can be parameterized.

2) ZMIPS *Assembler:* To enhance the expressiveness of ZMIPS, we further introduce the ability to define custom *Macros*, which are new user-defined instructions that are not part of the original ISA. In this case, the ZMIPS assembler treats a Macro as a sequence of existing instructions. The latter can improve usability and avoid repetition of instructions since functions and more complex constructions can now be defined as Macros.

Likewise, another assembler enhancement is support for custom *labels* in the code. Specifically, even though the abstract machine assembly instructions use *instruction numbers* as branch targets, the use of labels enables a convenient programming paradigm for users. At the assembler level, our labels are alphanumeric tags that begin and end by a double underscore (e.g., *__a_label__*), while inside Zilch these labels are converted to instruction numbers.

Finally, in our effort to make ZMIPS as compatible as possible with the MIPS ISA, we offer support for several assembler expressions, such as the text section (`.text`), and the data section (`.data`). Although these are not used by the Zilch abstract machine, their support renders the ZMIPS code backward compatible with MIPS simulators, save for the custom I/O instructions and absence of system calls.

E. Application Programming Interface (API) for Zilch

ZeroJava and ZMIPS assembly are powerful tools for developing new VC and ZKPK applications; however, additional attention is necessary for existing applications that rely on various system calls and standard library functions. Since our objective is to improve the usability of VC and ZKPK in a broad range of scenarios, Zilch further offers a convenient API that allows embedding computational

[4]MIPS invokes `syscall 9` to allocate heap memory. The number of bytes to allocate is passed to the `$a0` register, while `$v0` contains the address of the allocated memory.

integrity functionality into the code-base of existing C/C++ programs. Using our API, a programmer can independently invoke the prover and verifier of Zilch via C/C++ functions, where each invocation can support arbitrary functionality by passing a ZMIPS code snippet to the parent function. In effect, it is not necessary to convert an existing C/C++ application into ZeroJava/ZMIPS, except for the specific parts that require computational integrity. The next Section elaborates on our Zilch API, demonstrating two real-life case studies.

IV. Real Applications in Zilch

A. Vickrey Auction using Zilch API

To demonstrate the programming interface of Zilch, we implemented a Vickrey auction protocol (also known as *sealed-bid, second-price auction* [55]), in which bidders submit their private bids without knowing the bids of others. As in a traditional auction, the highest bidder wins, but the price paid equals the second-highest bid instead. In the Vickrey protocol, the auctioneer collects a bid and its cryptographic commitment from each bidder, and all commitments must satisfy two basic properties:

- **Binding:** For all non-uniform probabilistic polynomial-time algorithms, the probability of two messages m_1 and m_2 (where $m_1 \neq m_2$) will generate the same commitment c is negligible. Essentially, no bidder can find two different bids with the same commitment.
- **Hiding:** For all non-uniform probabilistic polynomial-time algorithms, the probability of extracting any information about the bid from its commitment is negligible. Hiding ensures that a bidder does not learn anything about the bids of others based on their commitments.

The binding and hiding requirements can be satisfied using a one-way collision-resistant hash function so that recovering a pre-image from the hash output or finding two pre-images with the same output would be intractable.

In our case study, we use the Davies-Meyer (D-M) one-way compression function and implement a single-block Merkle-Damgård hash construction [56] based on a block cipher $\mathcal{E}_k$; specifically, we employ the Speck cipher with 128 bits block-size and 128 bits key-size [57]. To construct a commitment $\mathcal{C}$, each bid value (up to 64 bits, zero extended) is concatenated with the bidder's commitment key (64 bits) and used as Speck's key input; the bidder's 128-bit random ID (rID) is used as the cipher input to be encrypted and also XORed with the resulting ciphertext, following the D-M construction [56]:

$$\mathcal{C} = rID \oplus \mathcal{E}_{key||bid}(rID) \tag{1}$$

For correct execution of the Vickrey scheme, although participants do not have knowledge about the bids of others, at the end of the auction each participant should be able to verify the correctness of the

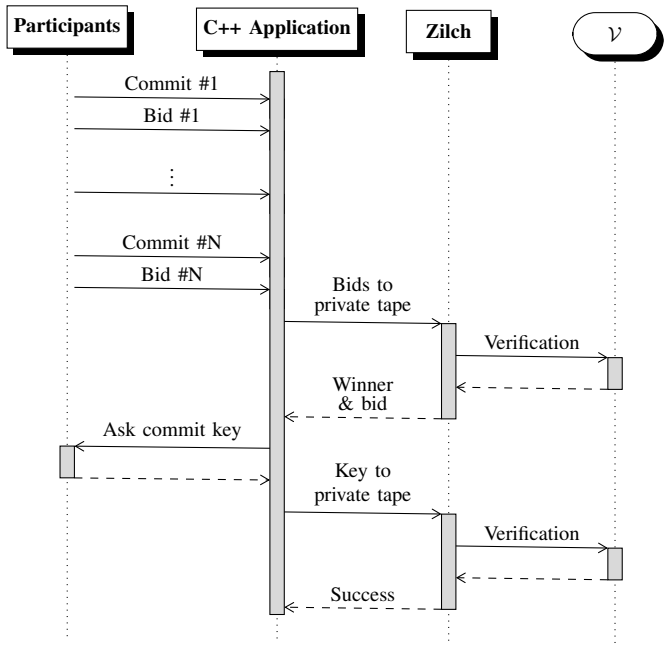


Fig. 5: Vickrey Auction Overview.

winning bid, even if the auctioneer is not entirely trusted (e.g., the auctioneer may be colluding with a bidder to increase the second-highest bid). Thus, a commitment scheme alone would not be sufficient and computational integrity is necessary to verify the correctness of the protocol.

We implement the auctioneer as a C++ application that collects the individual bids and hash commitments (Eq. 1) from all participants, before executing the Vickrey protocol to determine the winner and second-highest bid (Fig. 5). The C++ program employs our Zilch API to prove to each participant that the auctioneer function: (a) sorts all bids correctly to find the rID of the highest bidder, and (b) the highest bidder pays the second-highest bid. The latter requires proving computational integrity when the auctioneer opens the committed bids of the highest and second-highest bidders (i.e., verify Eq. 1 using $key||bid$ as the witness) and compares these bids with the announced second highest bid; the highest bidder should be convinced that the announced price was actually committed by someone, while the second-highest bidder should be convinced the there is someone that committed a higher bid. During this final step, the highest and second-highest bidder would send their commitment keys to the auctioneer. Overall, the auctioneer's code execution is verified and all bids remain private, except for the second-highest corresponding to the final price.

```
class RangeQuery {
  public static void main(String[] args) {
    int min, max, val;
    val = PrivateTape.read();
    min = PublicTape.read();
    max = PublicTape.read();
    if ((min <= val) && (val <= max)) {
      Prover.answer(true);
    }
    Prover.answer(false);
  }
}
```

Fig. 6: ZK range query implemented in ZeroJava.

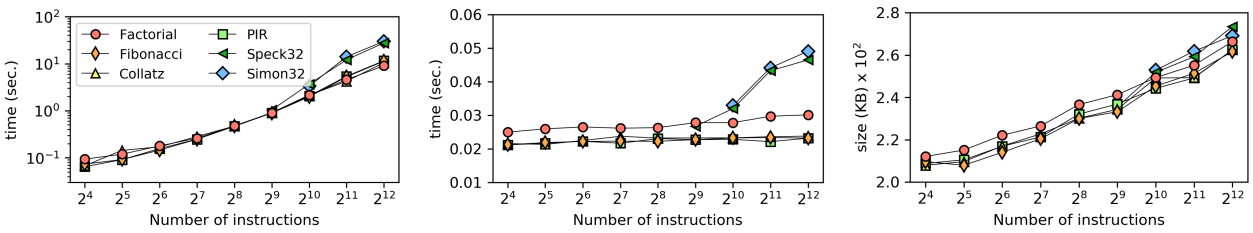
```
secread $t0          # read private input (val)
pubread $t1          # read min
pubread $t2          # read max
move $v0, 0          # result = false
bgt $t1, $t0, __end__    # if min > val
blt $t2, $t0, __end__    # if val < max
  move $v0, 1        # result = true
__end__:
answer $v0           # return result
```

Fig. 7: ZK range query implemented in zMIPS.

B. Zero-Knowledge Range Proofs with ZeroJava

Determining interest rates (e.g., when applying for a mortgage) may require disclosing the credit score of the applicant. Thus, another real-world application with Zilch would be to determine interest rates or loan eligibility while maintaining the privacy of credit scores. Likewise, Zilch can help proving that an account has enough available balance for a transaction, or that an individual is older than 18 years and younger than 65 years without disclosing the exact age. These examples belong to the broader class of zero-knowledge range proofs [58], where Zilch can verify that a secret number is within known bounds without actually disclosing it.

In Fig. 6 we illustrate the range query code implemented in ZeroJava, while Fig. 7 shows the compiled and optimized zMIPS assembly. Line 1 of the assembly reads the private value val (e.g., the age of an individual), while lines 2 and 3 read the lower (min) and the upper bound (max) from the public tape



(a) Prover time (b) Verifier time (c) Communication complexity size

Fig. 8: $\mathcal{P}$, $\mathcal{V}$ timings (seconds) and communication complexity size (KB) for a variety of benchmarks for different input sizes and 2^{-60} soundness error. The communication overhead corresponds to the interactive protocol between $\mathcal{P}$ and $\mathcal{V}$.

(e.g., ages 18 and 65 respectively). Consecutively, the program checks that val is within the given range (i.e., $min \leq val \leq max$) and returns either 0 or 1.

V. Experimental Evaluation

Experimental Setup: We implemented the ZeroJava compiler and optimizer in Java, while the rest of the Zilch framework is implemented in C++. We measured the runtime performance of Zilch using a variety of benchmarks described below. All experiments are obtained on a `t3.2xlarge` AWS EC2 instance running with eight virtual processors up to 2.5 GHz and 32 GB RAM on Ubuntu 20.04.

Multithreaded Prover: The back-end of the Zilch framework is highly parallelizable using OpenMP. It can take advantage of all available threads on the host, and we observe a 2x–4x speedup when using eight virtual cores on AWS.

A. Our Benchmarks

For our measurements, we adopt the TERMinator suite [59], which comprises scientific benchmarks designed for abstract machines like zMIPS. In particular, the TERMinator benchmarks are beneficial as they do not rely on OS features (such system calls) while covering a broad range of applications from kernel benchmarks to complex bit manipulations. For our analysis, we implemented the Speck and Simon lightweight block ciphers [57], where the former is oriented towards software implementations and the latter for circuit-based implementations: Speck is based on the Add-Rotate-XOR (ARX) paradigm, while Simon is a balanced Feistel cipher, and both support variable key and block sizes. Being symmetric encryption algorithms, Speck and Simon are very demanding in bitwise operations. Our evaluations also include the Factorial, Fibonacci, and Collatz sequences, as well as the matrix multiplication benchmark,

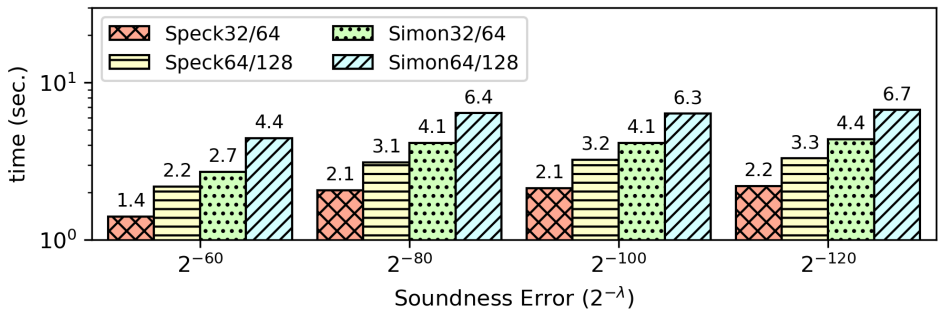


Fig. 9: $\mathcal{P}$'s measured execution time for the SPECK & SIMON cipher benchmarks using different security parameter sizes on the 32-bit and the 64-bit block sizes with 64-bit and 128-bit keys respectively.

all of which are addition and multiplication intensive. Moreover, a private information retrieval (PIR) program complements our set of benchmarks.

B. Experimental Results

In our evaluation, we assess the performance of Zilch on a variety of register word sizes (i.e., $W = 8, 16, 32$), as well as different soundness parameters (i.e., $\lambda = 60, 80, 100, 120$). For a soundness parameter λ, the probability that an untrusted prover would violate computational integrity and remain undetected is at most $2^{-\lambda}$. As expected and also confirmed in our benchmarks, larger values for W and λ increase the execution overhead for both $\mathcal{P}$ and $\mathcal{V}$.

In Fig. 8 we present the prover and verifier timings as well as the communication complexity sizes for the TERMinator benchmarks and how they scale with an increasing number of instructions (note, while ZMIPS is an abstract machine, its instructions are judiciously chosen to map to the MIPS ISA). For each benchmark, we vary the input size accordingly so that the total number of executed instructions matches a power of 2 and show how the prover and verifier timings depend on the number of instructions in the program. Fig. 8a shows quasi-linear prover overheads to the number of instructions ($T \cdot polylog(T)$), while SPECK and SIMON incur higher costs because bitwise operations require more complex constraints. Similarly, these two ciphers require poly-logarithmic ($polylog(T)$) verification time to the number of instructions, while the other benchmarks show constant overheads (Fig. 8b). Moreover, communication overheads increase linearly to the number of instructions (Fig. 8c).

Fig. 9 shows the prover's performance on SPECK and SIMON for key sizes 64-128 bits and varying security parameters. As expected, SPECK is faster than SIMON since the former has less instructions and is optimized for software. As λ grows larger, the proving time incurs higher overheads, yet, after 2^{-80} the impact is minimized: using SPECK32/64 as an example, an increase of λ from 60 to 80 adds

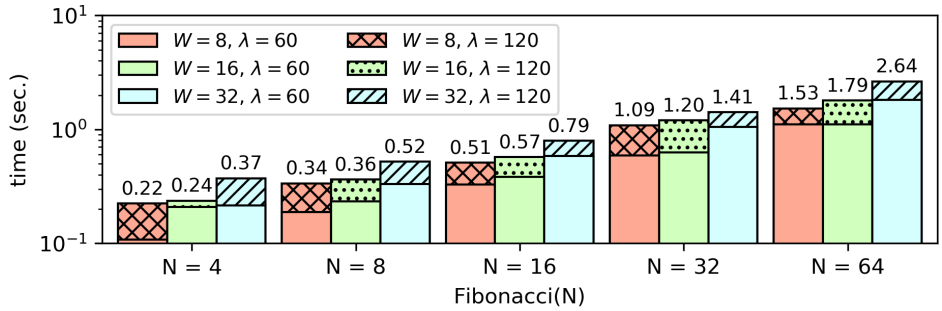
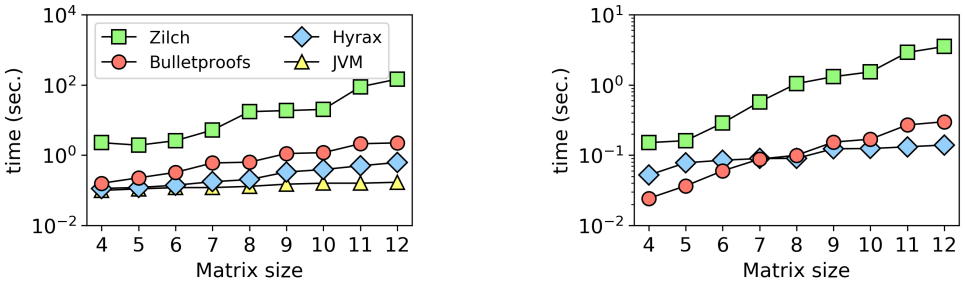


Fig. 10: $\mathcal{P}$'s measured execution time for the Fibonacci benchmark using different word-sizes $(8, 16, 32)$ and different security parameter sizes for a variety of inputs (2^2 to 2^6).

0.7 seconds to the proving time, whereas increasing λ from 80 to 120 adds only 0.1 seconds. Similar behavior is observed for both ciphers across all configuration sizes.

An overview of the runtime performance of our Fibonacci benchmark for different sizes of W and λ is presented in Fig. 10. The bars for $\lambda = 60$ are shown *in front* of those for $\lambda = 120$, and the exact values for the latter are reported. Our experiments show how performance overheads increase with both the input size N (as more instructions are required) and the wordsize W (as more complex constraints are required).



(a) Prover time

(b) Verifier time

Fig. 11: Comparison between Zilch, Hyrax and Bulletproofs $\mathcal{P}$ and $\mathcal{V}$ timings (seconds) for the matrix multiplication benchmark, as well as the native JVM baseline execution (i.e., without generating a proof).

C. Comparison with Previous Works

We compare Zilch with 80-bit security with two state-of-the-art transparent zero-knowledge systems: Hyrax [32] and Bulletproofs [30]. Both are based on elliptic curve cryptography and thus their security parameter is not directly comparable with Zilch's. We instantiated them using the M191 elliptic curve [60] over a base field modulo $2^{191} - 19$ giving approximately 90-bit security, based on their reference implementations [61].

For our analysis, we instantiated the standard SHA-256 hash algorithm and compare Zilch with Hyrax for an input block of 512 bits. Our results show that Zilch can prove the correct computation of one SHA-256 block in 73.86 seconds, while Hyrax requires 35.63 seconds; the $\mathcal{V}$ execution time was 1.55 and 1.19 seconds for Zilch and Hyrax respectively. Moreover, in Figs. 11a and 11b we report the $\mathcal{P}$ and $\mathcal{V}$ timings respectively using the matrix multiplication benchmark and matrix sizes varying from 4x4 to 12x12; as a baseline, Fig 11a also shows the native Java matrix multiplication cost (without any proof). Our results show that Hyrax has the fastest performance among the three systems, with Bulletproofs reporting similar timings; in comparison, the $\mathcal{P}$ and $\mathcal{V}$ cost of Zilch is almost one order of magnitude higher than Bulletproofs in matrix multiplication. Likewise, our comparison to the native Java execution shows the performance cost for proving computational integrity across the three systems.

Discussion: The main reason for the observed performance differences between Hyrax, Bulletproofs, and Zilch, is that the computations for the first two have been expressed directly as arithmetic circuits, whereas, in Zilch the computation was expressed in ZeroJava and then translated to zMIPS using our compiler. While operating directly on arithmetic circuits may achieve faster execution times, using a higher-level abstraction can significantly enhance usability. Notably, the authors' experience working with Hyrax and Bulletproofs showed that it is considerably hard, even for experienced programmers, to develop, debug and analyze any non-trivial program expressed using large monolithic ACs. This limitation informs why our comparisons with these related works focus on the two pre-compiled circuits already provided by these frameworks. As we discuss in Section VI, many related works rely solely on arithmetic circuits. Conversely, Zilch can easily be applied to any computation expressed in our high-level ZeroJava language.

Another important observation is that Zilch inherits from zk-STARK the property of plausible post-quantum security, which cannot be argued for either Hyrax and Bulletproofs. From a security perspective, since Zilch does not require any trusted setup and offers a broader threat model, it is not directly comparable with SNARK-based systems (e.g., [19], [21]–[23], [26]) that need a trusted setup; in fact, the total cost of having an offline trusted setup is not directly measurable, as it often includes expensive steps to eliminate the *toxic waste* (e.g., by physically destroying hard drives [62]). In Zilch, our goal is to move to a universal argument system that does not rely on trusted third parties and offers a usable programming model. In Section VI, we report further comparisons between Zilch's programming model and those of related works.

D. Zilch Experiments using our Real-life Case Studies

In this Section we evaluate the performance and programming complexity for the two real-life applications discussed in Section IV with security parameter $\lambda = 80$ and varying register sizes W.

Vickrey Auction: This application was developed in C++ and linked to Zilch using our C++ API; SPECK128/128 was developed in ZeroJava and compiled to ZMIPS instructions using the ZeroJava compiler. The word size used for SPECK128/128 is $W = 64$ bits, so both the 128-bit key and the 128-bit input block can fit in two registers each. Since this application is interactive across multiple participants, it entails multiple invocations of Zilch using our API (Fig. 5): The first invocation iterates on every bid stored in the private tape and performs comparisons to find the winner (highest bid) as well as the amount of the second-highest bid, while additional invocations are required to convince the first and second highest bidders. In Table IV we show how the $\mathcal{P}$ and $\mathcal{V}$ times depend on the total number of auction participants; the former is linear to the number of auction participants, while the latter is almost constant. In this case, since we rely on SPECK128/128 for computing each commitment $\mathcal{C}$ using in the D-M construction (Eq. 1), $\mathcal{P}$ performs a new evaluation of SPECK's key scheduling for each different $key||bid$ value of each participant. Each key scheduling requires about the same number of instruction as the SPECK core.

Zero-Knowledge Range Proofs: In our experiments, the high-level ZeroJava code for range-checking (Fig. 6) is compiled into ZMIPS instructions (Fig. 7) using our compiler. This example demonstrates how our programming paradigm in Zilch abstracts all low-level complications and programming complexity for ZKPs, enabling the programmer to express her intent using logical statements very similar to Java. With respect to performance, in this range-checking example, we measured less than 0.1 seconds of prover overhead and negligible verification time, using 16 and 32 bit register sizes.

TABLE IV: Vickrey auction: $\mathcal{P}$ and $\mathcal{V}$ times for increasing number of participants with security parameter $\lambda = 80$.

Participants	**Execution Steps**	$\mathcal{P}$ **Time (sec.)**	$\mathcal{V}$ **Time (sec.)**
8	71	0.37	0.025
16	151	1.96	0.026
32	311	4.15	0.026
64	631	8.67	0.027

VI. RELATED WORK

In the past few years, the interest of the academic community in VC and ZKPs was renewed, leveraging sophisticated cryptography, interactive and probabilistic checkable proofs. In this section, we discuss

TABLE V: Comparison of existing ZKP systems based on their cryptographic assumptions, the need for a trusted setup, their universality, and resilience against known attacks from quantum computers. Regarding *ease of programmability*, each bar indicates support for developing ZKPs using arithmetic circuits, assembly language, procedural and object-oriented programming, respectively. Among frameworks that support high-level programming (i.e., those with three or four bars), only Zilch supports the object-oriented paradigm.

Frameworks	**Protocol***	**Cryptographic Assumptions§**	**Transparent**	**Universal**	**Post-Quantum Resilient**	**Ease of Programmability ACs < ASM < PP < OOP†**	**Compiler Available**
Pinocchio [21]	zk-SNARK	KoE	○	○	○	▂▄▆▁	○
Geppetto [24]	zk-SNARK	KoE	○	○	○	▂▄▆▁	○
TinyRAM [22]	zk-SNARK	KoE	○	○	○	▂▄▆▁	○
Buffet [18]	zk-SNARK	KoE	○	○	○	▂▄▆▁	●
ZoKrates [63]	zk-SNARK	KoE	○	○	○	▂▄▆▁	●
xJsnark [64]	zk-SNARK	KoE	○	○	○	▂▄▆▁	●
vRAM [65]	zk-SNARG	KoE	○	●	○	▂▄▁▁	N/A
vnTinyRAM [23]	zk-SNARK	KoE	○	●	○	▂▄▆▁	○
MIRAGE [31]	zk-SNARK	GGM	○	●	○	▂▁▁▁	N/A
Sonic [35]	zk-SNARK	AGM	○	●	○	▂▁▁▁	N/A
Marlin [66]	zk-SNARK	KoE, AGM	○	●	○	▂▁▁▁	N/A
PLONK [67]	zk-SNARK	AGM	○	●	○	▂▁▁▁	N/A
SuperSonic [34]	zk-SNARK	ARA	●	●	○	▂▁▁▁	N/A
Bulletproofs [30]	zk-ShNARK‡	DL	●	●	○	▂▁▁▁	N/A
Hyrax [32]	zk-SNARK	DL	●	●	○	▂▁▁▁	N/A
Halo [68]	zk-SNARK	DL	●	●	○	▂▁▁▁	N/A
Virgo [33]	zk-VPD	CRHF	●	●	●	▂▁▁▁	N/A
Ligero [27]	zk-SNARK	CRHF	●	●	●	▂▁▁▁	N/A
Aurora [28]	zk-SNARK	CRHF	●	●	●	▂▁▁▁	N/A
zk-STARK [29]	zk-STARK	CRHF	●	●	●	▂▄▁▁	N/A
Zilch (this work)	zk-STARK	CRHF	●	●	●	▂▄▆▆	●

* SNARK stands for Succinct Non-Interactive ARgument of Knowledge, STARK stands for Scalable Transparent ARguments of Knowledge, SNARG stands for Succinct Non-interactive ARGuments, and VPD stands for Verifiable Polynomial Delegation.

§ KoE stands for Knowledge of Exponent, AGM stands for Algebraic Group Model, GGM stands for Generic Group Model, ARA stands for Adaptive Root Assumption, DL stands for Discrete Logarithm, and CRHF stands for Collision-Resistant Hash Functions.

† ACs stands for Arithmetic Circuits, ASM is Assembly language, PP is Procedural Programming, and OOP is Object-Oriented Programming.

‡ Bulletproofs is not considered a zk-SNARK because it is not succinct (i.e., has linear verification time). "Sh" stands for *short* instead of *succinct*.

several recent works in the area.

Trusted setup per computation: Gennaro et al. introduced in [17] quadratic arithmetic programs (QAP) which inspired many recent works such as Pinocchio [21] and other Succinct Non-Interactive Arguments of Knowledge (SNARKs) [18], [19], [24], [38], [41], [42], [69]–[71]. These protocols, in turn, formed the background for real-world systems as ZeroCash [72]. The proof size is in these constructions is succinct

and verification depends on the size of the argument being proven. However, contrary to Zilch, SNARKs require a trusted and expensive setup phase for every different statement. In many cases, real-world applications that require computational integrity cannot be founded on trusted third parties.

Universal trusted setup: Recent interactive proof-based techniques utilize universal and updatable trusted setups that are based on common – or structured – reference strings. Their advantage compared to the previous category is that they do not require a trusted pre-processing for each circuit, but only a single setup for all circuits. Such constructions include Sonic [35] that composes constant size proofs, as well as PLONK and Marlin [66], [67], which improve upon Sonic by constructing a different polynomial interactive oracle proof (IOP). Although these systems minimize the number of trusted setups to one, the random elements (toxic waste) that are used during this trusted phase may still be used by a malicious prover to forge proofs and break soundness.

Transparent setup: To address the previous limitations, various constructions emerged that are based on different cryptographic assumptions and do not require a trusted setup phase. Bulletproofs [30] and Halo [68] are based on the discrete logarithm problem, while other works such as Ligero [27], zk-STARKs [29], Aurora [28], and Virgo [33] leverage collision-resistant hashes, which can offer additional resilience against known attacks from quantum computers. Likewise, the work in [73] provides a construction that is secure in the quantum random oracle model. Other works such as [11] and [32] are based on interactive proofs. Finally, SuperSonic [34] proposes a new polynomial IOP that relies on groups of unknown orders and does not require a trusted setup. However, a notable limitation of the aforementioned systems is the lack of a practical programming model, which renders the development of ZKPs for arbitrary applications a daunting task.

Random Access Machines: The authors of [22] introduced a random-access machine targeting SNARKs called TinyRAM, which is based on a Harvard architecture. The work in [23] further introduced vnTinyRAM, which is a von Neumann alternative of the original TinyRAM. These TinyRAM variants, as well as vRAM [65], required a trusted pre-processing phase to generate parameters for verifying different arguments. Conversely, Zilch supports transparent setups where any required randomness is always public and can verify arbitrary programs for any given bound on the number of execution steps, leveraging the state-of-the-art zk-STARK library in its back-end. Notably, our ZMIPS ISA offers direct compatibility with existing MIPS programs and enables non-crypto-savvy programmers to easily develop *high-level object-oriented applications* for the ZMIPS abstract machine using our ZeroJava compiler and API, whereas this cannot be argued for other random access machines that build their own esoteric models. Lastly, ZMIPS supports special-purpose registers, labels, and user-defined Macros, rendering it a comprehensive ISA to ensure computational integrity in general-purpose computation.

Ease of Programmability: In Table V, we present comparisons between various zero-knowledge proof systems. Our comparisons are based on the requirement for a trusted setup, the universality of the ZKP system, the resilience to known attacks from quantum computers, and the ease of developing zero-knowledge proofs from a programmer's perspective.

Pinocchio [21], Geppetto [24], and Buffet [18] provide compilers for translating subsets of the C programming language to arithmetic circuits. In particular, these front-ends cover only a small subset of C (e.g., they do not support pointers) and also require the programmers to deviate from standard C code since they require defining extra constraints, casting statements, and prover-specific types. For example, loops can only have static termination conditions (i.e., cannot depend on non-constant variables) since they are unrolled by the compiler; in effect, this prevents having loops with early termination conditions and programmers must set a fixed bound for each loop. Moreover, while Buffet supports a somewhat larger subset of C compared to Pinocchio and Geppetto, it still lacks support for function pointers, `goto`, and loops with dynamic termination. Likewise, TinyRAM [22] relies on circuit representations that can be generated from C programs; however, the usability of this approach remains limited, as no compiler has been released, and the underlying ZKP protocol does not support a transparent setup. Similarly, ZoKrates [63] and xJsnark [64] provide front-ends to libSNARK [26] and enable programmers to express a computation using high-level programs that can be translated to arithmetic circuits. Their programming model, however, does not offer support for classes, inheritance, or polymorphism, contrary to the programming paradigm supported by Zilch. More importantly, as summarized in Table V, all aforementioned systems are based on zk-SNARKs and they require a new key generation phase to be invoked by a trusted third party *for each different computation* one wants to prove.

The works of vnTinyRAM [23] and vRAM [65] extend the random access machine introduced in [22] and enable universal circuits that can be used to verify any program up to a given number of machine steps without needing a new setup each time. Similarly, MIRAGE [31] proposes a universal circuit that consumes arithmetic circuits of a bounded number of operations as inputs. Nevertheless, while the generated circuits can implement arbitrary programs, all these works still require an initial trusted phase to set up the circuit. From a developer's perspective, vnTinyRAM employs the same C compiler that TinyRAM does; however, since no implementation of this compiler has been released, developers still have to resort to laborious assembly programming to express their algorithm.

Employing an orthogonal approach, recent zero-knowledge proof systems are also based on universal structured reference strings [35], [66], [67]. This approach allows a single trusted setup to support all circuits of some bounded size. Contrary to Zilch, these works can only support programming using arithmetic circuits and also require a setup phase by a trusted third party (i.e., they are not transparent).

The bottom section of Table V includes zero-knowledge proof systems that are transparent (like Zilch). Bulletproofs [30], Hyrax [32], and Halo [68] rely on the discrete logarithm problem, while SuperSonic [34] relies on groups of unknown order and the adaptive root assumption; as a result, these systems remain susceptible to attacks from quantum computers. Regarding their programming model, these proof systems support computations expressed as arithmetic circuits. Thus, to leverage these systems, a programmer has to manually implement the arithmetic circuit corresponding to their intended algorithm. While the released implementation of Hyrax [61] offers a set of custom scripts to automate this procedure for the arithmetic circuits of the matrix multiplication and SHA-256 examples, the development of new scripts remains as laborious as writing the arithmetic circuits directly. Contrary to Zilch, the programming model of Virgo [33], Ligero [27], Aurora [28] and zk-STARK [29] also relies on arithmetic circuits. While the reference implementation of zk-STARK offers partial support for TinyRAM instructions, however, critical operations such as reading private and public inputs are not supported [74]. Besides, zk-STARK does not offer any compiler to translate high-level programs into STARK proofs. Conversely, Zilch enables programmers to express a computation using our object-oriented ZeroJava language.

VII. Concluding Remarks

In this paper, we present Zilch, a framework to facilitate the deployment of verifiable computation and zero-knowledge proofs of knowledge for any application. Zilch is transparent (it does not rely on any trusted third party setup), post-quantum resilient, and using its easy-to-use programming model allows automated generation of universal circuits that can verify any arbitrary computation for a given time bound. In Zilch, we reduce the problem of proving arguments of knowledge to the granularity of an assembly instruction, so that we can verify instructions independently along with valid transitions between consecutive abstract machine states.

We have designed and implemented the zMIPS abstract machine, a MIPS-like processor model in which each instruction is intricately chosen and translated to a small arithmetic circuit. We complement our framework with a high-level language called ZeroJava and a compiler for translating ZeroJava code into optimized zMIPS assembly instructions. To further improve usability, we have defined a convenient programming API that allows integrating Zilch's prover and verifier into any existing C/C++ program. In our experiments, we demonstrate the performance of Zilch for a variety of benchmarks, as well as two real-life case studies.

References

[1] N. G. Tsoutsos and M. Maniatakos, "Fabrication attacks: Zero-overhead malicious modifications enabling modern microprocessor privilege escalation," *IEEE TETC*, vol. 2, no. 1, pp. 81–93, 2013.

[2] N. G. Tsoutsos, C. Konstantinou, and M. Maniatakos, "Advanced techniques for designing stealthy hardware trojans," in *DAC*. ACM, 2014, pp. 1–4.

[3] H. Takabi, J. B. Joshi, and G.-J. Ahn, "Security and privacy challenges in cloud computing environments," *IEEE S&P*, vol. 8, no. 6, pp. 24–31, 2010.

[4] P. Mahajan *et al.*, "Depot: Cloud storage with minimal trust," *ACM TOCS*, vol. 29, no. 4, p. 12, 2011.

[5] S. Arora *et al.*, "Proof verification and the hardness of approximation problems," *Journal of the ACM*, vol. 45, no. 3, pp. 501–555, 1998.

[6] S. Arora and S. Safra, "Probabilistic checking of proofs: A new characterization of NP," *Journal of the ACM (JACM)*, vol. 45, no. 1, pp. 70–122, 1998.

[7] L. Babai, "Trading group theory for randomness," in *Symposium on Theory of computing*. ACM, 1985, pp. 421–429.

[8] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," *SIAM Journal on computing*, vol. 18, no. 1, pp. 186–208, 1989.

[9] C. Lund *et al.*, "Algebraic methods for interactive proof systems," in *FOCS*. IEEE, 1990, pp. 2–10.

[10] A. Shamir, "IP = PSPACE (interactive proof = polynomial space)," in *FOCS*. IEEE, 1990, pp. 11–15.

[11] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, "Delegating computation: interactive proofs for muggles," *JACM*, vol. 62, no. 4, p. 27, 2015.

[12] G. Brassard, D. Chaum, and C. Crépeau, "Minimum disclosure proofs of knowledge," *Journal of Computer and System Sciences*, vol. 37, no. 2, pp. 156–189, 1988.

[13] J. Kilian, "A note on efficient zero-knowledge proofs and arguments," in *STOC*. ACM, 1992, pp. 723–732.

[14] S. Micali, "Computationally sound proofs," *SIAM Journal on Computing*, vol. 30, no. 4, pp. 1253–1298, 2000.

[15] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers," in *CRYPTO*. Springer, 2010, pp. 465–482.

[16] N. Bitansky *et al.*, "From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again," in *ITCS*. ACM, 2012, pp. 326–349.

[17] R. Gennaro *et al.*, "Quadratic span programs and succinct NIZKs without PCPs," in *Eurocrypt*. Springer, 2013, pp. 626–645.

[18] R. S. Wahby *et al.*, "Efficient RAM and control flow in verifiable outsourced computation." in *NDSS*, 2015.

[19] ——, "Verifiable ASICs," in *S&P*. IEEE, 2016, pp. 759–778.

[20] ——, "Full accounting for verifiable outsourcing," in *CCS*. ACM, 2017, pp. 2071–2086.

[21] B. Parno *et al.*, "Pinocchio: Nearly practical verifiable computation," in *S&P*. IEEE, 2013, pp. 238–252.

[22] E. Ben-Sasson *et al.*, "SNARKs for C: Verifying program executions succinctly and in zero knowledge," in *CRYPTO*. Springer, 2013, pp. 90–108.

[23] ——, "Succinct non-interactive zero knowledge for a von Neumann architecture," in *USENIX Security*, 2014, pp. 781–796.

[24] C. Costello *et al.*, "Geppetto: Versatile verifiable computation," in *S&P*. IEEE, 2015, pp. 253–270.

[25] D. Mouris and N. G. Tsoutsos, "Pythia: Intellectual Property Verification in Zero-Knowledge," in *DAC*. ACM/EDAC/IEEE, 2020, pp. 1–6.

[26] Succinct Computational Integrity and Privacy Research (SCIPR Lab), "libsnark," https://github.com/scipr-lab/libsnark, 2014, [Online].

[27] S. Ames *et al.*, "Ligero: Lightweight sublinear arguments without a trusted setup," in *CCS*. ACM, 2017, pp. 2087–2104.

[28] E. Ben-Sasson *et al.*, "Aurora: Transparent succinct arguments for R1CS," in *Eurocrypt*. Springer, 2019, pp. 103–128.

[29] ——, "Scalable zero knowledge with no trusted setup," in *CRYPTO*. Springer, 2019, pp. 701–732.

[30] B. Bünz *et al.*, "Bulletproofs: Short proofs for confidential transactions and more," in *S&P*. IEEE, 2018, pp. 315–334.

[31] A. Kosba *et al.*, "MIRAGE: Succinct Arguments for Randomized Algorithms with Applications to Universal zk-SNARKs," in *USENIX Security*, 2019.

[32] R. S. Wahby *et al.*, "Doubly-efficient zkSNARKs without trusted setup," in *S&P*. IEEE, 2018, pp. 926–943.

[33] J. Zhang *et al.*, "Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof," in *S&P*. IEEE, 2020.

[34] B. Bünz, B. Fisch, and A. Szepieniec, "Transparent snarks from dark compilers," in *Eurocrypt*. Springer, 2020, pp. 677–706.

[35] M. Maller *et al.*, "Sonic: Zero-knowledge snarks from linear-size universal and updatable structured reference strings," in *CCS*. ACM, 2019, pp. 2111–2128.

[36] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Eurocrypt*. Springer, 1986, pp. 186–194.

[37] M. Sipser *et al.*, *Introduction to the Theory of Computation*. Thomson Course Technology Boston, 2006, vol. 2.

[38] M. Walfish and A. J. Blumberg, "Verifying computations without reexecuting them," *Communications of the ACM*, vol. 58, no. 2, pp. 74–84, 2015.

[39] J. Groth, "Short pairing-based non-interactive zero-knowledge arguments," in *ASIACRYPT*. Springer, 2010, pp. 321–340.

[40] S. Setty, A. J. Blumberg, and M. Walfish, "Toward practical and unconditional verification of remote computations," in *USENIX HotOS*, vol. 13, 2011, pp. 29–29.

[41] J. Groth and M. Maller, "Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks," in *CRYPTO*. Springer, 2017, pp. 581–612.

[42] E. Ben-Sasson *et al.*, "Scalable zero knowledge via cycles of elliptic curves," *Algorithmica*, vol. 79, no. 4, pp. 1102–1160, 2017.

[43] J. Bootle *et al.*, "Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting," in *Eurocrypt*. Springer, 2016, pp. 327–357.

[44] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *FOCS*. Ieee, 1994, pp. 124–134.

[45] ——, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM review*, vol. 41, no. 2, pp. 303–332, 1999.

[46] J. Proos and C. Zalka, "Shor's Discrete Logarithm Quantum Algorithm for Elliptic Curves," *Quantum Info. Comput.*, vol. 3, no. 4, p. 317–344, Jul. 2003.

[47] D. J. Bernstein, "Introduction to post-quantum cryptography," in *Post-Quantum Cryptography*. Springer, 2009, pp. 1–14.

[48] P. Wegner, "A technique for counting ones in a binary computer," *Communications of the ACM*, vol. 3, no. 5, p. 322, 1960.

[49] S. D. Galbraith, C. Heneghan, and J. F. McKee, "Tunable balancing of RSA," in *Australasian Conference on Information Security and Privacy*. Springer, 2005, pp. 280–292.

[50] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, principles, techniques," *Addison wesley*, vol. 7, no. 8, p. 9, 1986.

[51] B. Bishop and F. Fischer, "Iris-integrated rule inference system," in *International Workshop on Advancing Reasoning on the Web: Scalability and Commonsense (ARea 2008)*. sn, 2008.

[52] Y. Smaragdakis and M. Bravenboer, "Using Datalog for fast and easy program analysis," in *International Datalog 2.0 Workshop*. Springer, 2010, pp. 245–251.

[53] E. Ben-Sasson *et al.*, "Computational integrity with a public random string from quasi-linear PCPs," in *Eurocrypt*. Springer, 2017, pp. 551–579.

[54] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Newnes, 2013.

[55] W. Vickrey, "Optimal auctions," *The American Economic Review*, vol. 71, no. 3, pp. 381–392, 1981.

[56] J.-S. Coron *et al.*, "Merkle-damgård revisited: How to construct a hash function," in *CRYPTO*. Springer, 2005, pp. 430–448.

[57] R. Beaulieu *et al.*, "The SIMON and SPECK lightweight block ciphers," in *DAC*. ACM/EDAC/IEEE, 2015, pp. 1–6.

[58] T. Koens, C. Ramaekers, and C. Van Wijk, "Efficient Zero-Knowledge Range Proofs in Ethereum," Technical Report, Tech. Rep., 2018.

[59] D. Mouris, N. G. Tsoutsos, and M. Maniatakos, "TERMinator Suite: Benchmarking Privacy-Preserving Architectures," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 122–125, 2018.

[60] D. F. Aranha *et al.*, "A note on high-security general-purpose elliptic curves." Cryptology ePrint Archive, Report 2013/647, 2013.

[61] R. S. Wahby *et al.*, "Reference implementation of Hyrax and Bulletproofs," https://github.com/hyraxZK/hyraxZK, 2018, [Online].

[62] ZCash, "Parameter Generation Ceremony and Destruction of Toxic Waste," https://z.cash/technology/paramgen/, 2016.

[63] J. Eberhardt and S. Tai, "ZoKrates - Scalable Privacy-Preserving Off-Chain Computations," in *iThings/GreenCom/CPSCom/SmartData*. IEEE, 2018, pp. 1084–1091.

[64] A. Kosba, C. Papamanthou, and E. Shi, "xJsnark: a framework for efficient verifiable computation," in *S&P*. IEEE, 2018, pp. 944–961.

[65] Y. Zhang *et al.*, "vRAM: Faster verifiable RAM with program-independent preprocessing," in *S&P*. IEEE, 2018, pp. 908–925.

[66] A. Chiesa *et al.*, "Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS," in *Eurocrypt*. Springer, 2020, pp. 738–768.

[67] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge," Cryptology ePrint Archive, Report 2019/953, 2019.

[68] S. Bowe, J. Grigg, and D. Hopwood, "Halo: Recursive Proof Composition without a Trusted Setup," Cryptology ePrint Archive, Report 2019/1021, 2019.

[69] M. Backes, D. Fiore, and R. M. Reischuk, "Verifiable delegation of computation on outsourced data," in *CCS*. ACM, 2013, pp. 863–874.

[70] A. E. Kosba *et al.*, "TRUESET: Faster Verifiable Set Computations," in *USENIX Security*, 2014, pp. 765–780.

[71] M. Backes *et al.*, "ADSNARK: nearly practical and privacy-preserving proofs on authenticated data," in *S&P*. IEEE, 2015, pp. 271–286.

[72] E. B. Sasson *et al.*, "Zerocash: Decentralized anonymous payments from bitcoin," in *S&P*. IEEE, 2014, pp. 459–474.

[73] A. Chiesa, P. Manohar, and N. Spooner, "Succinct arguments in the quantum random oracle model," in *TCC*. Springer, 2019, pp. 1–29.

[74] E. Ben-Sasson *et al.*, "libSTARK: a C++ library for zk-STARK systems," https://github.com/elibensasson/libSTARK, 2018, [Online].