Zero Knowledge for Everything and Everyone:
Fast ZK Processor with Cached RAM for ANSI `C` Programs

David Heath, heath.davidanthony@gatech.edu
Yibin Yang, yyang811@gatech.edu
David Devecsery, ddevec@fb.com
Vladimir Kolesnikov, kolesnikov@gatech.edu

Abstract

We build a complete and efficient ZK toolchain that handles proof statements encoded as arbitrary ANSI `C` programs.

Zero-Knowledge (ZK) proofs are foundational in cryptography. Recent ZK research has focused intensely on non-interactive proofs of small statements, useful in blockchain scenarios. We instead target large statements that are useful, e.g., in proving properties of programs.

Recent work (Heath and Kolesnikov, CCS 2020 [HK20a]) designed a proof-of-concept ZK machine (ZKM). Their machine executes arbitrary programs over a minimal instruction set, authenticating in ZK the program execution. In this work, we significantly extend this research thrust, both in terms of efficiency and generality. Our contributions include:

- A rich and performance-oriented architecture for representing arbitrary ZK proofs as programs.
- A complete compiler toolchain providing *full support for ANSI* `C95` *programs.* We ran off-the-shelf buggy versions of `sed` and `gzip`, proving in ZK that each program has a bug. To our knowledge, this is the first ZK system capable of executing standard Linux programs.
- Improved ZK RAM. [HK20a] introduced an efficient ZK-specific RAM BubbleRAM that consumes $O(\log^2 n)$ communication per access. We extend BubbleRAM with *multi-level caching*, decreasing communication to $O(\log n)$ per access. This introduces the possibility of a cache miss, which we handle cheaply. Our experiments show that cache misses are rare; in isolation, i.e., ignoring other processor costs, BubbleCache improves communication over BubbleRAM by more than $8\times$. Using BubbleCache improves our processor's total communication (including costs of cache misses) by $\approx$ 25-30%.
- Numerous low-level optimizations, resulting in a CPU that is both more expressive and $\approx 5.5\times$ faster than [HK20a]'s.
- Attention to user experience. Our engineer-facing ZK instrumentation and extensions are minimal and easy to use.

Put together, our system is efficient and general, and can run many standard Linux programs. The resultant machine runs at up to 11KHz on a 1Gbps LAN and supports MBs of RAM.

1 Introduction

Zero Knowledge (ZK) protocols allow a prover $\mathcal{P}$ to demonstrate to a verifier $\mathcal{V}$ the truth of a given statement while revealing nothing additional. ZK proofs (ZKPs) are foundational cryptographic objects and have attracted wide research attention. ZK research originally focused on proofs of specific statements, but more recent works developed efficient techniques that handle proofs of *arbitrary* statements. Even with the pivot towards generality, the community's focus has remained on succinct proofs of small statements, motivated largely by blockchain applications. In contrast, proofs of large, complex statements have been left relatively unexplored.

Our focus: ZK for large programs. Recent work showed that ZK protocols can support a relatively efficient ZK *processor* [HK20a]. [HK20a] designed a new arithmetic ZK protocol and data structures, including a ZK-specific RAM BubbleRAM. They integrated these components into a processor that executes arbitrary programs in ZK, supports hundreds of KB of memory, and runs at 2.1KHz over a small instruction set. We continue [HK20a]'s research thrust into large proof statement support by improving both the ZK processor's efficiency and the tooling available to programmers.

1.1 Our Contribution

We propose the first complete and concretely efficient ZK system for general `C` programs and `libc`[1]. Our system allows $\mathcal{P}$ and $\mathcal{V}$ to run a mutually agreed `C` program describing a proof statement via an interactive constant-round ZKP protocol. Our contribution comprises intertwined crypto-technical, architectural, and systems-technical components. We present:

- *An architecture for representing ZK proofs as programs.* While circuit-based ZK is efficient even for large circuits (e.g., [HK20b, WYKW20]), circuits do not scale to programs with complex control flow. We design a complete architecture for high-performance CPU-emulation-based ZK, carefully balancing the cost and expressivity of each CPU step. Our architecture includes ZK-specific primitives; most notably, our *prover oracle calls* allow the programmer to effectively handle $\mathcal{P}$'s input and to shortcut expensive algorithms that can be efficiently verified. We compare with previous ZK architectures [HK20a], [BCG+13] in Section 2.

- *Extensive compiler and library support.* We fully support ANSI `C95` programs and `libc`.

- *Improved ZK RAM.* ZK RAM allows the proof to efficiently look up an element from memory obliviously to $\mathcal{V}$. [HK20a] introduced an efficient ZK-specific RAM BubbleRAM that costs $O(\log^2 n)$ communication per access. We extend BubbleRAM with multi-level caching, decreasing

[1] Both [HK20a] and [BCG+13] lack `libc` and cannot support off-the-shelf programs. They run, respectively, $\approx 5.5\times$ and $10000\times$ slower than our work.

communication to $O(\log n)$ per access. We call our caching RAM BubbleCache. BubbleCache introduces the possibility of a cache miss, which we handle cheaply. Our experiments show that cache misses are rare, and, ignoring other processor costs, BubbleCache improves communication over BubbleRAM by more than $8\times$. Using BubbleCache improves our processor's total communication (including costs of cache misses) by $\approx$ 25-30%.

- *Numerous low-level crypto optimizations* as compared to [HK20a], such as a greatly improved ALU, tighter integration of instructions into the CPU circuit, and an efficient small table lookup technique. In sum, our non-memory based operations consume only 86 oblivious transfers, about a $4\times$ improvement over [HK20a]. We also extend [HK20a]'s protocol with generalized vector-scalar multiplication.

- *User experience (UX).* Our goal is to make ZK easy to use by engineers who are not trained in cryptography. We adhere to standard `C`, including native Linux I/O, user-input, and file-system operations, and require only that the programmer includes special `QED` instruction(s) as appropriate: reaching `QED` means a successful proof.

- *Use-cases drawn from existing popular Linux programs.* We ran off-the-shelf, buggy versions of `gzip` and `sed`, proving in ZK the existence of bugs (CVE-2005-1228 and the SIR repository, respectively). The `gzip` bug consumes $44,092$ CPU cycles and runs in 6.5s; the `sed` bug consumes $390,002$ CPU cycles and runs in 36.1s. Each benchmark is $\approx$ 5.4KLOC of `C` code.

- *We plan to open-source* our project to the community.

2 Related Work

We compile programs written in high level `C` down to a low-level instruction set, and we evaluate instructions on an emulated ZK CPU. Our technique allows proofs of arbitrary statements, including statements that are large and complex. In our review of related work, we focus on concretely efficient ZK protocols and on works that pursue secure CPU-emulation.

[HK20a] is the most relevant work, and we build on many of its techniques. Section 5 reviews their ZK protocol, which we build on. In this section, we compare to [HK20a] in several contexts: architecture, RAM, performance, support for general `C`. In sum, our work is better on all of these fronts. We highlight our cached RAM, our ability to run standard `C` programs, and our $\approx 5.5\times$ clock rate improvement.

ZK ZK proofs [GMR85, GMW91] allow $\mathcal{P}$ to convince $\mathcal{V}$, holding circuit C, that there exists an input, or *witness*, w for which $C(w) = 1$. Early practical

ZK protocols, motivated by signatures and identification schemes, focused on algebraic relations. More recently, ZK research has shifted focus to proofs of arbitrary statements.

ZK from garbled circuits (GC-ZK) and 2PC GC-ZK techniques encode ZK relations as garbled circuits [JKO13, FNO15, HK20b]. [JKO13] established a Garbling Scheme (GS)-based ZKP *framework*: by satisfying a few requirements, a new GS can be plugged into [JKO13]'s protocol to obtain malicious-verifier ZK. The [HK20a] protocol, which we leverage, is formalized in the [JKO13] framework.

[HK20b] showed that GC-ZK can efficiently handle *conditional branching*. Their 'stacked garbling' technique yields communication that scales only with the longest execution path, not with the size of the circuit. While [HK20b] improves communication, their *computation* remains linear in the number of branches. Natural circuit-based handling of arbitrary control flow can result in significant blow-up in circuit size, and is often infeasible. Our work also scales only in the longest execution path, but does so by arranging the program into CPU steps. Because we change the program representation, we enjoy *both* communication and computation that scale linearly in the program's execution time. We did not apply stacked garbling in our design. Our CPU is hand-optimized to aggressively amortize low-level operations. Porting stacking to the [HK20a] protocol is not trivial and would not substantially improve our already lean CPU.

Wolverine [WYKW20] recently improved over GC-based ZK by instead running a maliciously-secure GMW protocol where $\mathcal{V}$ has no input, allowing all GMW gates to be run in parallel. We use [HK20a]'s protocol which similarly runs all gates in parallel: our prover $\mathcal{P}$ simultaneously requests wire labels on each algebraic wire and then proves they are all related. In terms of cost, [HK20a]'s protocol is quite similar to Wolverine "per gate" for our desired security and representation. Wolverine requires 2–4 field elements per arithmetic gate in a large field. To compute our basic operation, vector-scalar multiplication of a vector with n field elements, we perform a single 1-out-of-2 OT of n field elements. Thus both we and Wolverine transmit a similar number of field elements per arithmetic gate.

Unlike direct circuit representations, CPU-based techniques allow arbitrary control flow and scale to more realistic programs. Of course, CPU-based ZK is built on circuit protocols, so circuit protocol improvements remain an excellent direction for future work.

ZK Processors and Architectures [BCTV14b, BCTV14a, BCG+13] implement ZK processors via succinct non-interactive proof engines. In a sense, these approaches are more general than ours: our approach is interactive and requires linear communication. The trade-off is efficiency. These works yield processors that run in the 1Hz range. In contrast, our processor operates in the 10KHz range. These works also introduced the TinyRAM architecture, which

is similar in scope to our architecture: it allows proofs of arbitrary statements formulated as `C` programs. We build a custom architecture so as to extract as much performance as possible from the underlying protocol. As a result, our non-RAM components are cheap. TinyRAM CPU (without RAM operations) consumes ≈ 1000 algebraic constraints per cycle, while ours consumes only 86 OTs (OTs are comparable to constraints).

[HK20a] implements a concretely efficient interactive ZK processor and is the closest to our work. Their focus is proof-of-concept, so they only support a small subset of `C`. In contrast, we comprehensively handle `C` programs.

Our architecture provides ZK-specific features supported neither by [HK20a] nor by [BCG+13]. Most notably, we include instructions that we call prover oracle calls. Prover oracle calls allow the programmer to structure $\mathcal{P}$'s input and also to use $\mathcal{P}$ to shortcut algorithms that are expensive to compute but efficient to verify. We view prover oracle calls as a crucial component for general ZK handling. Spice [SAGL18] used remote procedure calls (RPC), a mechanism similar to our prover oracle calls, to achieve similar goals.

We mention, but do not discuss in detail, MPC processors [LO13, WGMK16, SHS+15]; their task is different and they are much less efficient than our ZK processor.

Succinct and non-interactive ZK Our work is built on an efficient interactive protocol with proof size linear in the program running time. Other works emphasize non-interactivity and/or succinct proofs. We review such works in Appendix A. In short, these works have excellent performance for small proof statements, but they struggle when handling larger proof statements.

ZK RAM A key component in our processor is our ZK BubbleCache, an improvement on the state-of-the-art BubbleRAM [HK20a]. In our experiments (Section 9), BubbleCache is up to $8\times$ more efficient than BubbleRAM (improvement depends on RAM size), which was already concretely efficient. Asymptotically, BubbleCache improves BubbleRAM's $O(\log^2 n)$ cost to $O(\log n)$, although it introduces the possibility of cache misses. Like BubbleRAM, BubbleCache does not use extra rounds of communication.

[BCG+13] and a subsequent work [WSR+15] gave RAM-like constructions whereby $\mathcal{P}$ provides all RAM values as inputs and then, at the end of the protocol, permutes all read/write values and demonstrates consistency. The technique features high concrete costs: (1) $\mathcal{P}$ provides each value as extra input and (2) the consistency check requires checking that values are appropriately sorted. In contrast, BubbleRAM/BubbleCache read values directly from storage and feature an *extremely* cheap consistency check: on each access, $\mathcal{P}$ simply proves two values are equal, requiring amortized zero bytes of communication. More specifically, for a RAM of size n and for k RAM operations, Buffet's [WSR+15] cost is $(21 + 10\log k + 2\log n)$ arithmetic constraints per RAM operation. BubbleRAM incurs $\frac{1}{2}\log^2 n$ constraints (i.e., OTs) per RAM operation, and BubbleCache further improves on BubbleRAM as explained above.

Spice [SAGL18] specified an $O(1)$ cost key-value store. Their approach elegantly improves over Merkle-tree based RAMs by maintaining two efficiently updatable hash digests: one for all reads and one for all writes. On an access, $\mathcal{P}$ provides the requested element as input, and the proof accordingly updates the two hashes. At the end of the proof, the two hashes are used to 'audit' all accesses, demonstrating that $\mathcal{P}$ honestly constructed each input. While the approach incurs amortized $O(1)$ cost per access, the concrete costs are relatively high: their approach (and straightforward $O(\log n)$ Merkle-tree based ZK RAMs) require that the hash function be evaluated *inside* the ZK circuit. BubbleCache does not use non-black-box hash function calls.

3 Notation and Assumptions

3.0.1 Notation

- We refer to our running system as ZKM: ZK machine.
- σ is the statistical security parameter (e.g., 40).
- κ is the computational security parameter (e.g., 128).
- The prover is $\mathcal{P}$. We refer to $\mathcal{P}$ by she, her, hers...
- The verifier is $\mathcal{V}$. We refer to $\mathcal{V}$ by he, him, his...
- We write $\triangleq$ to denote that the left hand side is *defined* to be the right hand side.
- We denote that x is uniformly drawn from a set S by $x \in_{\$} S$.
- We denote the field of elements $\{0, 1, ..., p-1\}$ by $\mathbb{Z}_p$.
- We denote the set of elements $\{1, 2, ..., p-1\}$ by $\mathbb{Z}_p^{\times}$.
- We work with additive secret shares in a field $\mathbb{Z}_p$. We denote a sharing of a semantic value $x \in \mathbb{Z}_p$ by $[\![x]\!]$. Sharings are formally defined in Section 5.

3.0.2 Cryptographic Assumptions

We use the recent and efficient Ferret oblivious transfer technique [YWL+20]. Thus, we inherit its assumptions: (1) learning parity with noise (LPN), (2) a tweakable correlation-robust hash function, and (3) a random oracle (RO). If we assume access to an OT oracle, our technique is secure under standard assumptions.

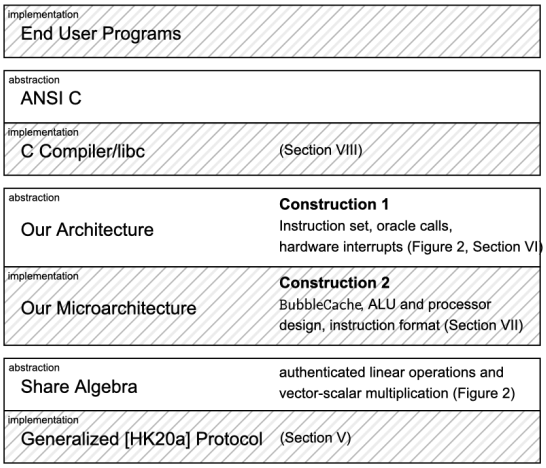

Figure 1: The stack of abstractions and implementations in our system. Each implementation is built using the abstraction below it. At the top level, users write custom ANSI `C` programs. We implement ANSI `C` and `libc` via our custom compiler and runtime library. Our compiler interfaces with our architecture, which is the high level design of our emulated processor; the architecture consists primarily of our instruction set. We refer to the implementation of our architecture as our *microarchitecture*. The microarchitecture consists of low level processor components and algorithms, such as our ALU and BubbleCache. Finally, our microarchitecture is implemented in terms of authenticated shares, realized by [HK20a]'s ZK protocol.

4 Presentation Roadmap

At a high level, we bridge end user programs and an underlying ZK protocol. We achieve this in two steps (see Figure 1). First, end user proof statements are expressed in high level `C`. These `C` programs are translated into ZK assembly by our custom compiler toolchain. The resultant assembly is specified by our *architecture*, the high level abstraction of our emulated processor. Second, we execute the assembly on our custom emulated processor. Our *microarchitecture* (the implementation of our architecture) consists of a number of low-level and algorithmic choices that emphasize efficiency. The microarchitecture is implemented on top of authenticated shares, provided by our generalization of [HK20a]'s protocol.

Our presentation proceeds bottom-up through Figure 1's abstraction stack, starting from the most primitive cryptographic objects and working up to the handling of end user programs:

- Section 5 reviews [HK20a]'s authenticated share algebra as well as their cryptographic protocol.
- Section 6 presents our architecture, including a high level description of our machine as well as its instruction set. We also present extended discussions of so-called prover oracle calls and of hardware interrupts.
- Section 7 describes our microarchitecture with an emphasis on algorithmic improvements, including our caching ORAM BubbleCache and our efficient ALU.
- Section 8 shows how we integrated our processor into a user-facing system. We describe our `C` compiler and `libc` implementations, and we show how the parts of our system interoperate.

The cross-cutting concern of the stack in Figure 1 is performance: many choices in our design are ultimately informed by the cost of operations in the authenticated share algebra.

Section 9 concludes with experimental findings of the efficiency of our system.

5 [HK20a]'s Authenticated Share Algebra

Our processor is built on [HK20a]'s ZK protocol. [HK20a]'s protocol implements the primitives listed in Figure 2.

In the protocol, $\mathcal{P}$ and $\mathcal{V}$ hold *authenticated sharings* of values in a field $\mathbb{Z}_p$ for a suitably large prime p (we choose $p = 2^{40} - 87$, the largest 40 bit prime). An authenticated sharing consists of two *shares*, one held by $\mathcal{V}$ and one by $\mathcal{P}$. We denote a sharing where $\mathcal{V}$'s share is $s \in \mathbb{Z}_p$ and $\mathcal{P}$'s share is $t \in \mathbb{Z}_p$ by writing $\langle s, t \rangle$. At the start of the protocol, $\mathcal{V}$ samples a non-zero global value $\Delta \in_\$ \mathbb{Z}_p^\times$. Consider a sharing $\langle X, x\Delta - X \rangle$ where $X \in_\$ \mathbb{Z}_p$ is drawn by $\mathcal{V}$. A sharing of

	Operation	Communication (Bytes)
Primitive	$[\![x]\!] + [\![y]\!] \mapsto [\![x+y]\!]$	0
	$[\![x]\!] - [\![y]\!] \mapsto [\![x-y]\!]$	0
	$c[\![x]\!] \mapsto [\![cx]\!]$	0
	$c \mapsto [\![c]\!]$	0
	$x[\![\vec{y}]\!] \mapsto [\![x\vec{y}]\!]$	$5 \cdot \lvert\vec{y}\rvert$
	$\mathcal{P}$ proves $[\![x]\!] = [\![0]\!]$	0 (amortized)
Composite	$x \mapsto [\![x]\!]$	5
	Decompose 32-bit word into bits	$32 \cdot 5 = 160$
	Permute n m-word elements	$5 \cdot m \cdot n \cdot \log n$

Figure 2: ZK protocol operations and their communication cost. Primitive operations include (1) adding two sharings, (2) subtracting two sharings, (3) multiplying a sharing by a public constant, (4) encoding a public constant as a sharing, (5) multiplying a vector of sharings by a bit selected by $\mathcal{P}$, and (6) checking that a sharing encodes zero. Above, c denotes a public constant and x outside of a share denotes a value chosen by $\mathcal{P}$. Complex operations are composed from primitives; we list common composite operations and their costs.

this form is a *valid* sharing of the semantic value $x \in \mathbb{Z}_p$. We use the shorthand $[\![x]\!]$ to denote a valid sharing:

$$[\![x]\!] \triangleq \langle X, x\Delta - X \rangle \qquad \text{where } X \in_\$ \mathbb{Z}_p$$

Sharings have two key properties:

1. $\mathcal{V}$'s share gives no information about the semantic value. This holds trivially: $\mathcal{V}$'s share is independent of x.

2. $\mathcal{P}$'s share is 'unforgeable': $\mathcal{P}$ cannot use $x\Delta - X$ to construct $y\Delta - X$ for $y \neq x$. This is because (1) both X and Δ are uniform and unknown to $\mathcal{P}$, (2) the multiples of Δ are uniformly distributed over the field, and (3) the chosen prime is large enough to achieve our desired security: $\mathcal{P}$ can forge $[\![y]\!]$ only by guessing $y\Delta - X$, which only succeeds with probability $\frac{1}{p-1}$.

We review primitive operations on shares in Appendix B, including our simple but very useful generalization to the protocol: a vector-scalar multiplication that generalizes [HK20a]'s share multiplication technique. The interface to and cost of protocol primitives are given in Figure 2. In short, linear operations over sharings are computed locally, but each vector-scalar multiplication requires communication in the form of a single *oblivious transfer* (OT).

6 An Architecture for ZK Programs

This section presents our ZK architecture. Specifically, we formalize our processor and its instruction set.

We express ZK relations as high level C programs. Section 2 discussed that circuit-based ZK dominates in the literature, but that circuits do not scale to arbitrary control flow. We instead adopt a *CPU emulation-*, or *ZK Processor*-based architecture: $\mathcal{P}$ and $\mathcal{V}$ jointly, *authentically and obliviously to* $\mathcal{V}$ execute a sequence of CPU steps that together evaluate the program. $\mathcal{V}$ accepts the proof if the program terminates in some distinguished state. Each CPU step is implemented by a circuit; we carefully manage these circuits to ensure correct execution of the program.

Section 6.1 begins informally, summarizing the "platform" on which we build our architecture. Section 6.1 may be seen as crypto background for an architecture person: it presents costs of basic functions and derives informal guidelines for efficient operation. Then, we formalize our architecture in Sections 6.2 and 6.3, giving the description of our ZK machine as well as its instructions. Sections 6.4 and 6.5 then proceed in detail on the more interesting parts of our architecture: prover oracle calls and hardware interrupts.

6.1 ZK Processor Intuition and Efficiency Guidelines

Ultimately, our ZK processor is built on simple algebra implemented by an efficient ZK protocol. While our current goal is to construct an abstract architecture, we do so keeping in mind the concrete efficiency properties that our implementation ultimately inherits. Thus, we informally discuss basic primitives and pieces of intuition underlying our extensions and improvements over [HK20a], on which we build. In particular, we derive basic *guidelines* for processor design. Costs discussed in this section are based on concrete choices of security parameters: 128-bit computational security and 40-bit statistical security.

As a reminder, the protocol allows $\mathcal{P}$ and $\mathcal{V}$ to operate on *authenticated sharings* of semantic values that appear during the processor execution. Sharings have two crucial properties:

1. Sharings are *oblivious*, meaning that they convey no information about the semantic values to $\mathcal{V}$.

2. Sharings are *authentic*, meaning that it is infeasible for $\mathcal{P}$ to create a share that is not computed by a procedure agreed upon by both $\mathcal{V}$ and $\mathcal{P}$.

6.1.1 Protocol Primitives

The protocol allows $\mathcal{P}$ and $\mathcal{V}$ to manipulate integers in the field $\mathbb{Z}_p$ for a large prime p. The protocol's primitive operations over sharings include addition, subtraction, and a form of vector-scalar multiplication (see Figure 2 and Appendix B). This vector-scalar multiplication works as follows: (1) $\mathcal{P}$ holds a private scalar $x \in \{0, 1\}$, (2) the parties input a shared vector $[\![\vec{y}]\!]$, and (3) the parties output the shared scaled vector $[\![x\vec{y}]\!]$.

These primitives provide a starting point for implementing the entire processor state machine, including reading and decoding instructions, reading/writing memory, and operating over processor registers. While addition and subtraction

are in a sense 'free' (i.e. require no communication), vector-scalar multiplication requires the parties to interact via oblivious transfer (OT).[2] OT cost includes fixed overhead: executing its basic component Random OT requires either (1) transmission of 16 bytes (κ bits) with "traditional" techniques or (2) significant computation with the recent Silent OT works [BCG+19, YWL+20]. OT also includes variable overhead: 5 bytes (σ bits) are sent per vector element. It is the handling and transmission of these OTs which bottleneck performance, so optimizing the processor mostly involves decreasing the processor's *multiplicative complexity*. Thus, we arrive at our first guideline.

Guideline 1. *Minimize multiplications.*

We realize Guideline 1 by aggressively amortizing vector-scalar multiplication. This amortization takes two forms:

1. We seek optimizations that amortize fixed OT overhead by favoring small numbers of multiplications of long vectors over large numbers of multiplications of short vectors. One extreme example is given in Section 7.3, where we implement small table lookup using a logarithmic number of vector-scalar multiplications.

2. We seek optimizations that re-use the same non-linear operations to compute different values, perhaps with the help of additional linear operations. As a simple example, our processor computes both bit-wise AND and bit-wise OR each cycle. Rather than computing both operations separately, we use multiplication to compute only bit-wise AND, and then use linear operations to derive bit-wise OR from the result.[3]

6.1.2 Data movement

A significant portion of the work done by our processor involves conditionally moving information from one place to another, most notably in the processor's RAM. Conditionally moving data can be implemented by multiplication. While the available multiplication primitive places a constraint on the scalar x (x must be 0 or 1), the vector $\vec{y}$ is unconstrained (each index can hold an arbitrary $\mathbb{Z}_p$ element). Thus, whether we store individual bits in $\vec{y}$ or if we store entire *words* in $\vec{y}$, the cost of multiplication *remains the same*. This leads to our second guideline:

Guideline 2. *Favor moving data as words rather than as bits.*

Our processor manipulates 32-bit words. Guideline 2 instructs us to represent all 32 bits as a single sharing rather than separately sharing each bit, reducing the cost of data movements by 32×. As an additional benefit, the parties store data more cheaply: if parties store data as shares of bits, they incur

[2] OT is a foundational cryptographic primitive that allows a receiver R to learn one of the sender S's two secrets (1) without R learning the other secret and (2) without S learning R's choice.

[3] Let $a, b \in \{0, 1\}$. Then $a \wedge b = ab$ and $a \vee b = a + b - ab$.

40× storage blow-up. Word-based storage requires only 40 bits of storage per player per word.

6.1.3 Oblivious RAM

RAM is a tricky subject in ZK: on each RAM access, $\mathcal{V}$ must not learn the queried index. The typical strategy for implementing such an *oblivious RAM* (ORAM) involves carefully permuting data such that is impossible for $\mathcal{V}$ to observe an access pattern. We can permute a large array by performing a large number of small data movements via a construction called a Waksman permutation network [Wak68]. Unfortunately, permuting n elements requires $O(n \log n)$ vector-scalar multiplications:

Guideline 3. *Avoid costly permutations.*

To this end, we make use of [HK20a]'s BubbleRAM construction, a ZK specific ORAM that carefully amortizes permutations. We take this yet a step further by augmenting BubbleRAM with a caching mechanism, a trick that reduces the amortized RAM lookup cost from $O(\log^2 n)$ to $O(\log n)$ (see Section 7.1). While our improved BubbleCache is much faster than BubbleRAM per access, it introduces the possibility of cache misses. In practice, we have found that cache misses are rare enough that their possibility is more than made up for by decreased access cost (cf section 9).

6.1.4 Mixing Boolean and arithmetic

Guideline 2, which instructs us to store data in words, has a downside: we often need to perform bitwise operations, such as AND or OR. Such operations require each of the value's bits to be represented by a distinct sharing. In order to both follow Guideline 2 and allow bitwise operations, we need a mechanism that decomposes integer values into their constituent bits. Bit decomposition can be built with the help of two protocol primitives: (1) a primitive that allows $\mathcal{V}$ and $\mathcal{P}$ to convert a constant c to a sharing $[\![c]\!]$ and (2) a primitive that allows $\mathcal{P}$ to prove to $\mathcal{V}$ that a sharing holds zero. Note, $\mathcal{P}$ can convert an input bit to a sharing: if $\mathcal{P}$ has a bit of input $x \in \{0, 1\}$, the parties multiply $x \cdot [\![1]\!]$. Now, we can implement bit decomposition:

$$\text{decompose} : \mathbb{Z}_{2^{32}} \to \{0, 1\}^{32}$$

The procedure is as follows: (1) $\mathcal{P}$ separately inputs each output bit (she knows the output bits in cleartext), (2) the parties use addition to compose the bits into a word, and (3) $\mathcal{P}$ proves that the composed word is equal to the input word by proving their difference is zero. This last step proves that $\mathcal{P}$'s provided bit decomposition is valid, so the parties output those shares. Each decomposition requires $\mathcal{P}$ to input 32 bits, a relatively expensive operation:

Guideline 4. *Avoid bit decomposition.*

Our implementation performs exactly two bit decompositions every cycle: in general, each algebraic instruction has two input registers, and we decompose the values in both registers. Moreover, we amortize bit decomposition multiplications with those needed by other operations (per Guideline 1). The two 32 bit decompositions, requiring 64 OTs, are the most expensive subcomponent of our processor's ALU.

The above guidelines inform many of our architectural decisions, and we refer to them to motivate our choices.

6.2 High Level Architecture

We now formalize our architecture.

Construction 1 (Architecture)**.** *Our Architecture is the abstract machine described in this section and in Section 6.3.*

Values manipulated by our architecture are 32-bit integer words. Construction 1 consists of the following components:

- A 32-bit program counter `pc`.
- An instruction memory $\mathcal{I}$. Instructions are stored separately from data in a read-only memory (ROM).[4] Separating instructions allows us to perform fewer permutation operations than if we had stored them together with data (Guideline 3), particularly because ROMs can be efficiently implemented in ZK [HK20a]. Note, the program (i.e., the content of the ROM) is publicly agreed on by $\mathcal{P}$ and $\mathcal{V}$, though the access order must be kept secret.
- A *registry* $\mathcal{R}$ with 32 registers that each hold a single word. Instructions operate directly on the registers.
- A *main memory* $\mathcal{M}$ with 2^k words of memory for custom k. Memory is *word-addressable* (as opposed to byte-addressable, see Guidelines 2 and 4). Specific `LOAD` and `STORE` instructions move data between $\mathcal{R}$ and $\mathcal{M}$.
- An *input tape*, holding $\mathcal{P}$'s input values.
- A publicly agreed upon running time $\mathcal{T}$.

At initialization, (1) `pc` is set to zero, (2) each word in $\mathcal{R}$ and $\mathcal{M}$ are set to zero, (3) $\mathcal{I}$ is loaded with the program, and (4) $\mathcal{T}$ is chosen. The processor then executes $\mathcal{T}$ *cycles*. On each cycle, the processor reads instruction $\mathcal{I}[\texttt{pc}]$ and accordingly performs a combination of the following:

[4] Thus our architecture is a 'Harvard architecture', as opposed to a more typical 'von Neumann architecture' in which instructions and data are stored together. When proving the existence of program bugs, it is possible that a Harvard architecture could exhibit bugs differently than a von Neumann architecture. We did not observe this as a practical issue when replicating bugs - see Section 9.2.

1. Load up to two values from $\mathcal{R}$.
2. Load a value from $\mathcal{M}$.
3. Load a value from the processor's input tape.
4. Perform an ALU operation on loaded values.
5. Store a value in $\mathcal{R}$.
6. Store a value in $\mathcal{M}$.
7. Update `pc`.

The specific actions corresponding to each instruction are given in Section 6.3. After $\mathcal{T}$ cycles, the machine loads one final instruction: if $\mathcal{I}[\texttt{pc}]$ holds the distinguished instruction `QED`, then the machine outputs 1 (i.e., $\mathcal{V}$ accepts the proof) and otherwise outputs 0 (i.e., reject).

Our architecture also includes *hardware interrupts*: on an illegal memory access, the processor jumps to a distinguished program location. Section 6.5 explains our interrupts.

6.3 Our Instruction Set

Figure 3 enumerates the syntax and semantics of the twenty instructions in our architecture. We organize our instructions into four general categories: (1) instructions that perform simple algebra over the registers, (2) instructions that allow arbitrary control flow, (3) instructions that manipulate main memory, and (4) instructions that collect $\mathcal{P}$'s input. We mention several interesting instructions:

- `QED` is our distinguished 'proof' instruction. $\mathcal{P}$ succeeds if she provides input that leads to `QED`.
- `HALT` is our distinguished 'fail' instruction: since the processor is stuck on `HALT`, it is impossible to subsequently reach `QED` and complete a proof.
- `ORACLE` manages $\mathcal{P}$'s input if $\mathcal{P}$ is honest. Section 6.4 discusses `ORACLE` at length.
- `PC` provides the minimal functionality required for handling procedure calls: before jumping to a procedure, the processor must store `pc` such that it can properly return.

Our instruction set carefully balances expressivity with each CPU step's *multiplicative complexity* (see Guideline 1). Looking forward to our microarchitecture, our ALU, which implements our algebraic instructions, can be computed using only 65 OTs because of careful amortization (see Section 7.2). Other candidate instructions, such as a bitwise left shift, would not fully amortize and

	Syntax	Semantics
Algebra	`MOV` tar $\{src\}$	$\mathcal{R}[tar] \leftarrow val(src)$
	`CMOV` tar src_0 $\{src_1\}$	$\mathcal{R}[tar] \leftarrow \begin{cases} val(src_1), & \text{if } \mathcal{R}[src_0] \neq 0 \\ \mathcal{R}[tar], & \text{otherwise} \end{cases}$
	`ADD` tar src_0 $\{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] + val(src_1))$
	`SUB` tar src_0 $\{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] - val(src_1)$
	`MUL` tar src_0 $\{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] \cdot val(src_1)$
	`XOR` tar src_0 $\{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] \oplus val(src_1)$
	`AND` tar src_0 $\{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] \wedge val(src_1)$
	`OR` tar src_0 $\{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] \vee val(src_1)$
	`EQZ` tar src	$\mathcal{R}[tar] \leftarrow \begin{cases} 1, & \text{if } \mathcal{R}[src] = 0 \\ 0, & \text{otherwise} \end{cases}$
	`MSB` tar src	$\mathcal{R}[tar] \leftarrow \begin{cases} 1, & \text{if } \mathcal{R}[src] \geq 2^{31} \\ 0, & \text{otherwise} \end{cases}$
	`POW2` tar src	$\mathcal{R}[tar] \leftarrow 2^{\mathcal{R}[src]}$
Control	`JMP` $\{dst\}$	$\texttt{pc} \leftarrow val(dst)$
	`BNZ` src $\{dst\}$	$\texttt{pc} \leftarrow \begin{cases} val(dst), & \text{if } \mathcal{R}[src] \neq 0 \\ \texttt{pc} + 1, & \text{otherwise} \end{cases}$
	`PC` tar $\{src\}$	$\mathcal{R}[tar] \leftarrow \texttt{pc} + val(src)$; $\texttt{pc} \leftarrow \texttt{pc} + 1$
	`HALT`	– no effect, `pc` unchanged –
	`QED`	– no effect, `pc` unchanged –
Memory	`LOAD` tar a_0 $\{a_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{M}[\mathcal{R}[a_0] + val(a_1)]$
	`STORE` src a_0 $\{a_1\}$	$\mathcal{M}[\mathcal{R}[a_0] + val(a_1)] \leftarrow \mathcal{R}(src)$
$\mathcal{P}$ Input	`INPUT` tar	$\mathcal{R}[tar] \leftarrow x$ where $x \in \{0..2^{32}-1\}$ from $\mathcal{P}$
	`ORACLE` $\{id\}$	$\mathcal{P}$ calls procedure $val(id)$; $\texttt{pc} \leftarrow \texttt{pc} + 1$

$$val(x) \triangleq \begin{cases} x, & \text{if } x \text{ is an immediate} \\ \mathcal{R}[x], & \text{if } x \text{ is a register id} \end{cases}$$

Figure 3: Our instruction set. Each instruction type handles between zero and three arguments. In general, arguments refer to *registers*, but some arguments, denoted $\{\cdot\}$, can also optionally be *immediates* (i.e., compile-time constants). *val* is a helper function that resolves an argument that can be either a register or an immediate. Unless the semantics otherwise mention an effect on the `pc`, each instruction also increments the `pc`.

would thus require additional OTs. Since left shift is easily implemented by two instructions (`POW2` and `MUL`), we omitted this instruction[5] and others.

6.4 $\mathcal{P}$'s Input and Prover Oracle Calls

Cryptographers often view the problem of arranging $\mathcal{P}$'s input as outside the concern of ZK: $\mathcal{P}$ "just knows" her input. In practice, such omniscience must be arranged: $\mathcal{P}$'s input can be arbitrarily complex. We believe that, to provide a suitable interface, a ZK architecture should cleanly integrate the handling of $\mathcal{P}$'s input. Our architecture provides this integration via a concept called a *prover oracle call*.

Prover oracle calls (or just oracle calls) are initiated by our `ORACLE` instruction. $\mathcal{P}$ subsequently (1) exits the ZK proof, (2) computes in cleartext an arbitrary procedure over the current processor state, (3) loads values onto the input tape, and (4) re-enters the proof. With the input tape properly loaded, the ZK program can issue `INPUT` instructions to word-by-word read from the tape.

$\mathcal{P}$ maintains a cleartext *table* of oracle call procedures. At runtime, she interprets the `ORACLE` instruction's argument as an index, looks up the corresponding procedure, and runs that procedure on the current processor state. We provide a suite of useful oracle calls, and the programmer can extend the table by dynamically linking custom code.

At a high level, we use oracle calls for two primary tasks.

First, we use oracle calls to interface with $\mathcal{P}$'s cleartext system. E.g., in our `libc` implementation, when the ZK machine reads a file, $\mathcal{P}$ escapes the ZK proof, reads from the target file on her local machine, and loads the resulting bytes onto the input tape. The ZK machine then loads the bytes off the tape via `INPUT`. See Section 8 for more discussion.

Second, we use oracle calls to shortcut procedures that are expensive to compute but efficient to verify. Given a function f with input x, it is not necessary for $\mathcal{P}$ and $\mathcal{V}$ to compute $f(x)$ under the ZK protocol. Many ZK protocols take advantage of this fact. We make this capability available to end users via oracle calls: $\mathcal{P}$ computes locally and loads $f(x)$ onto the input tape (this clairvoyance is why we call them 'oracle calls'). Now, the parties need only verify that x and $f(x)$ are related by f. Verification is sometimes far easier than evaluation. E.g., while sorting a list requires $O(n \log n)$ operations, verifying that a given permutation sorts a list uses only $O(n)$ operations (see Section 9.4). As another example, bitwise right shift is expensive to compute directly with our instruction set, but can be easily verified using bitwise operations and multiplication.

After a 'shortcut' oracle call is complete, it is *essential* that $\mathcal{P}$'s inputs be verified: a malicious $\mathcal{P}$ can provide arbitrary inputs rather than following the specification of the oracle call.

Because we provide `ORACLE` as an *instruction*, such behaviors can be changed and extended by the programmer. This said, we view oracle calls as a feature

[5]The key difference between `POW2` and left shift is that it is efficient to ensure that `POW2` will not overflow the prime field.

primarily for 'power users'; users new to ZK can write ordinary `C` without using the feature or even knowing that it exists.

6.5 Hardware Interrupts

Programs can enter inconsistent states, such as when dereferencing an invalid memory location. Such errors cannot be handled directly by the program itself. Computer systems address such scenarios via *hardware interrupts*, special function calls performed at the time of fault. We introduce interrupts in our architecture as well. In computer systems, as well as in ZK, interrupts can be used more generally: for example, for event- and timer-based interrupts.

Currently, our architecture only issues an interrupt on an illegal memory access. In this case, the processor sets `pc` to a statically determined, constant value `MEMFAULT`. By default, index `MEMFAULT` holds the `HALT` instruction; a program that issues an illegal memory access is by default a failed proof. However, the code at `MEMFAULT` can be customized. This is particularly useful for proving the existence of bugs: $\mathcal{P}$ might wish to demonstrate that a particular input will cause a program to access invalid memory. Rather than hard-coding this behavior, we take a more general approach: the code at `MEMFAULT` can be set to `QED`, or to any arbitrary program.

As a final detail, we mark a protected region in the middle of the memory space that, upon access, triggers an interrupt. This ensures separation of the stack and the heap, and is a standard security technique used in cleartext systems.

7 Our Microarchitecture

We now formalize our *microarchitecture*, i.e. the implementation of our architecture.

Construction 2. *Our Microarchitecture is the concrete implementation of the Architecture (Construction 1) described in this section.*

Our microarchitecture manipulates authenticated sharings of values in the field $\mathbb{Z}_p$ for a given prime p (concretely, we choose $p = 2^{40} - 87$). Construction 2 instantiates the architecture components as follows:

- The program counter $\texttt{pc} \in \mathbb{Z}_p$ is a sharing $[\![\texttt{pc}]\!]$.
- The instruction memory $\mathcal{I}$ is implemented by an efficient ZK ROM as specified by [HK20a]. The ROM consumes $O(\log n)$ OTs per instruction lookup. Our physical instruction format is described in Section 7.4.
- The registry $\mathcal{R}$ is implemented by BubbleCache (see Section 7.1) with 32 slots.
- The main memory $\mathcal{M}$ is implemented by BubbleCache with 2^k slots for publicly agreed k.

- The input tape is implemented by allowing $\mathcal{P}$ to provide a 32-bit value on every cycle. We emphasize that the input tape is *not concretely represented* inside ZK. However, if $\mathcal{P}$ is honest and runs our implementation (note, there is no requirement that she does: a malicious $\mathcal{P}$ can run arbitrary software), she maintains a cleartext input tape that maintains her input choices.

- The number of time steps $\mathcal{T}$ is instantiated by a publicly agreed cleartext integer. $\mathcal{P}$ does not use the number of time steps prescribed by our *architecture*: she must also pad $\mathcal{T}$ to account for cache misses.

At initialization, `pc`, $\mathcal{R}$, and $\mathcal{M}$ are zero-initialized using the protocol's support for constants. Similarly, $\mathcal{I}$ is initialized with the constant program text.

The processor performs $\mathcal{T}$ cycles. At the start of each cycle, $\mathcal{P}$ inputs a single bit that allows her to *skip* the current cycle, including updating the `pc`. This capability allows $\mathcal{P}$ to skip cycles when the processor would otherwise incur a cache miss (see Section 7.1). On a skipped cycle, RAM continues to make progress towards a cache hit.

On each cycle, the processor loads instruction $\mathcal{I}[\texttt{pc}]$ and performs *all* of the following actions:

- Load two values from $\mathcal{R}$.

- Load a value from $\mathcal{M}$.

- Read an input from $\mathcal{P}$.

- Based on the instruction, multiplex (1) the second register value, (2) the loaded memory value, (3) the instruction immediate, and (4) $\mathcal{P}$'s input. This computes the second loaded value (the first is the first loaded register value).

- Compute *all* ALU operations on the two loaded values (see Section 7.2).

- Compute *all* possible `pc` updates.

- Multiplex the ALU output and `pc` update based on the instruction opcode.

- Compute mod 2^{32} on the ALU output, ensuring that the ALU output (which is a $\mathbb{Z}_p$ element) fits in the set $\mathbb{Z}_{2^{32}}$.

Finally, the processor performs *all* of the following:

- Update a value in $\mathcal{R}$.

- Update a value in $\mathcal{M}$.

- Update `pc`.

In many cases and depending on the instruction, each 'update' may leave the target untouched.

We describe many of the details above in the following sections. At a high level, our microarchitecture is the relatively straight-forward implementation of our architecture, albeit with many optimizations. Our key claim is the following:

Theorem 1. *Assuming a collision resistant hash function, that the prime modulus $p > 2^{37}$, and that $\lfloor \log p \rfloor \geq \sigma$, our microarchitecture (Construction 2) is a* sound *(with soundness error $2^{-\sigma}$) and* complete *malicious-verifier Zero Knowledge machine that proves arbitrary ZK relations expressed in our ISA (Construction 1) in the OT-hybrid model.*

The proof of this theorem relies on the JKO framework [JKO13] and follows naturally (1) from discussion throughout this section, (2) from the security of the [HK20a] protocol, and (3) from the fact that we check $\mathcal{P}$'s input values where appropriate. We defer a more formal treatment of the proof of Theorem 1 to Appendix D.

We use $\sigma = 40$; increasing σ will lead to a linear cost increase.

7.1 BubbleCache

[HK20a] introduced BubbleRAM, a ZK-specific ORAM with high concrete performance. For a RAM with n elements, BubbleRAM consumes only $\frac{1}{2}\log^2 n$ OTs per access. On each access, BubbleRAM allows $\mathcal{P}$ to shuffle a portion of the RAM. By doing so she ensures that, on each access, the needed element is in the zeroth slot of RAM. $\mathcal{P}$ proves that her shuffles are correct by proving that each requested index is equal to the index stored in RAM. Permutations of large arrays are expensive to compute: BubbleRAM achieves each permutation via a Waksman permutation network [Wak68], a circuit that consumes $n \log n$ OTs. However, BubbleRAM is carefully scheduled such that large permutations are infrequent. Thus, permutation cost is amortized across accesses.

BubbleRAM's design ensures that RAM is strictly correct: the parties are guaranteed to successfully look up the requested index. In this section, we show that BubbleRAM can be significantly improved by removing strictness. At a high level, we modify BubbleRAM such that memory is shuffled less often. Our modification, which we call BubbleCache, must therefore allow for *cache misses*. In practice, the frequency of cache misses depends on how aggressively we reduce shuffling and on the proof statement. While the frequency of shuffling can be chosen arbitrarily, we elect a strategy that results in $O(\log n)$ OTs per access both because of the asymptotic improvement and because it is effective in practice.

7.1.1 BubbleRAM formal review

BubbleRAM's key primitives are oblivious *partitions*. A partition on an array of n elements allows $\mathcal{P}$ to select $\frac{n}{2}$ elements and to permute the array such that those $\frac{n}{2}$ elements are in the first $\frac{n}{2}$ array slots; the remaining $\frac{n}{2}$ elements are

placed into the last $\frac{n}{2}$ slots. Aside from partitioning these two halves of the array, the partition guarantees nothing about the relative order of the elements. An oblivious partition on n elements can be achieved by $O(n \log n)$ OTs via a Waksman permutation network.

Let N be the number of elements in RAM. BubbleRAM maintains a time-counter t. t is initialized to zero; after each access, t is incremented. The key idea is to carefully partition memory on a schedule according to t. On each access, BubbleRAM considers each power of two $2^k \leq N$ starting from the largest such k and working down to $k = 1$. BubbleRAM allows $\mathcal{P}$ to partition the first 2^k elements if $2^{k-1} \mid t$, that is if 2^{k-1} divides t. This strategy is based on the following argument. If $\mathcal{P}$ places the next $\frac{n}{2}$ needed elements into the first $\frac{n}{2}$ slots, then we need not look at the last $\frac{n}{2}$ slots for the next $\frac{n}{2}$ accesses: it is impossible that a needed element would be in the back of the array. However, once we have exhausted $\frac{n}{2}$ accesses, $\mathcal{P}$ must repartition memory to again account for the fact that needed elements might be at the back of RAM. BubbleRAM stores each RAM element in a tuple together with its index, allowing the CPU to efficiently check that $\mathcal{P}$ properly moves each needed element to slot zero.

7.1.2 Adding Caching to BubbleRAM

It is widely understood that, in practice, programs exhibit high *data locality*: once an element is used, it is likely that the same element will be needed again soon. As described so far, BubbleRAM ignores this fact. To take advantage of data locality, we will allow BubbleRAM to "miss" from time to time. We use this allowance to "spread out" the BubbleRAM schedule, reducing the number of partitions and hence the overall cost: we can partition the first 2^k elements when $f(k) \mid t$ for arbitrary f.[6] By varying f, we can trade between RAM performance (a sparse schedule is cheap) and cache miss-rate (in a sparse schedule, it is likely $\mathcal{P}$ will be unable to put the next needed element in slot zero at the correct time). By spreading out the schedule, we essentially transform BubbleRAM from an efficient array to an even more efficient memory space with $\log N$ levels of cache. Each cache level can be given its own schedule, allowing us to tune cost trade-offs at each level.

Our processor accounts for cache misses by allowing $\mathcal{P}$ to inject no-ops: $\mathcal{P}$ predicts if the RAM will miss and, if so, instructs the processor to skip a cycle. By doing so, she skips BubbleRAM's index comparison check, which she would otherwise certainly fail because the needed element is not in slot zero. During the skipped cycle, the RAM continues to increment its time counter and perform partitions, so the skipped cycle makes progress towards a cache hit. This preserves ZK, since no-op cycles are indistinguishable to $\mathcal{V}$.

Cache misses fit well in our CPU emulation-based architecture where no-ops can be easily injected. The same cannot be said of direct circuit representation-based architectures, where each potential miss must be conditionally handled.

[6] Or we can use any Boolean valued function as our scheduler. We focus on *periodic* schedules because they are natural and effective.

This added handling further exacerbates the already challenging problem of circuit-based control flow.

The value of cache-based memory is well understood in the world of cleartext computing. Our specific choice of f is loosely based on cleartext caching where each progressively larger level of cache is more expensive than the last, mapping to the idea that the more recently a piece of data was used, the more likely it will be needed again soon. One curious point is that, in terms of miss-rate, we outperform cleartext RAM since $\mathcal{P}$ has the rather unique opportunity to use Belady's optimal page replacement algorithm to optimally program partitions [Bel66].

7.1.3 BubbleCache

We now present our specific choice of schedule. Consider the function $\uparrow$ which, for an input x, computes the nearest power of two greater than x:

$$x \uparrow \triangleq 2^{\lceil \log x \rceil}$$

Let C be a constant. We use the following scheduler:

$$(C \cdot k \cdot 2^{k-1}) \uparrow \mid t \tag{1}$$

That is, we delay partitioning by factor $C \cdot k$ (we vary C in Section 9; values near $\frac{1}{2}$ yield high performance). The function $\uparrow$ ensures that the partition period for level i divides the period for each level $j > i$. This prevents partitions at large levels of cache from 'ruining' partitions at low levels of cache (recall, we do not assume anything about the relative ordering within the two halves of a partition): at the time we partition a large level, we will subsequently partition all smaller levels.

Construction 3. *BubbleCache is BubbleRAM whose scheduler is configured according to Equation* (1)*.*

It is difficult to argue *analytically* that logarithmic delay is a good choice: data locality depends on the application and the input, and so is highly variable. We choose logarithmic delay for two reasons: (1) it yields excellent per-access performance both asymptotically (see Theorem 2) and concretely and (2) in our experiments it yields low cache miss-rates. More efficient caching algorithms are possible, especially if we are willing to base the caching strategy on the target program. However, any improvements are likely to be small since BubbleCache has only $\log N$ cost with small constants.

Theorem 2 (BubbleCache efficiency)**.** *BubbleCache consumes $O(\log N)$ OTs per access.*

Proof. By amortizing partition costs across accesses.

Consider a single element as it moves through the levels of cache and eventually ends up in slot zero. That element's arrival in slot zero is arranged by $\log N$

partitions, one for each power of two less than N. Each partition of $n = 2^k$ elements consumes $O(n \log n)$ OTs and is amortized over $(C \cdot k \cdot 2^{k-1}) \uparrow$ accesses. Thus, we summarize the cost of a single access:

$$O\left(\sum_{k=1}^{\log N} \frac{2^k \cdot \log(2^k)}{(C \cdot k \cdot 2^{k-1}) \uparrow}\right) = O\left(\sum_{k=1}^{\log N} \frac{k \cdot 2^k}{k \cdot 2^k}\right) = O(\log N)$$

Thus, each access consumes $O(\log N)$ OTs. □

We use BubbleCache to implement both our main memory and our registry. The caching mechanism greatly decreases the communication cost of each processor cycle, and we have found that the cache miss-rate is generally low.

7.2 ALU: Efficient Arithmetic

Our processor implements arithmetic instructions via an arithmetic logic unit (ALU). We defer a formal specification of the ALU to Appendix C. Here, we note interesting points:

Our construction uses only a 40-bit prime field, as compared to the 64-bit field used by [HK20a]. This is a significant improvement: the sizes of messages transmitted by OTs are reduced. The key reason [HK20a] chose a 64-bit prime is because naïve multiplication of 32-bit numbers can yield a value as high as $(2^{32} - 1)^2$. A 64-bit prime field ensures that even this worst-case multiplication will not overflow. We designed a form of "bit-wise" multiplication that outputs a maximum value $32 \cdot (2^{32} - 1) < 2^{37}$: hence our choice of a 40-bit prime field is informed by security, not by correctness.

Our ALU aggressively amortizes multiplications: the ALU, which handles all arithmetic instructions, needs only 65 OTs. Our ALU significantly contributes to our overall performance.

Our architecture operates over 32-bit integers, but our protocol only supports a prime field. Thus, we must *embed* 32-bit integers in the field. We achieve this by computing mod 2^{32} on each ALU output. We do so cheaply: after the ALU runs, $\mathcal{P}$ provides input that subtracts the top bits off an overflowing value before we store the result in the output register. Although $\mathcal{P}$ chooses this input freely, this trick is automatically secure: our ALU always decomposes its arguments into 32 bits, and the validity of the decomposition is checked. This check passes only if $\mathcal{P}$ honestly subtracted off the top bits appropriately. This does mean a malicious $\mathcal{P}$ can store a non-32-bit value in the registry, but she cannot use this method to cheat: she cannot read a bad value from the registry without being caught.

7.3 Multiplexing: Efficient Small Table Lookup

Each cycle has two primary tasks: (1) updating the program counter and (2) updating the output register (storing to main memory is also an important effect, but is handled separately). Each of the twenty instructions in our ISA

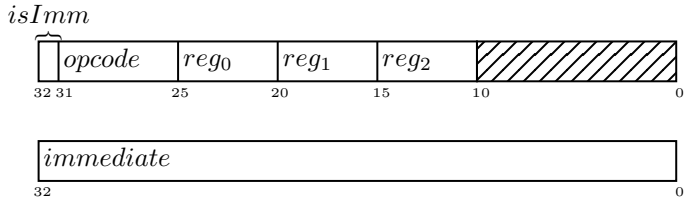

Figure 4: Our instruction physical format. Each instruction has six fields: (1) a single bit $isImm$ that indicates if the instruction uses an immediate, (2) the instruction's op-code, (3–5) up to three register arguments, and (6) the full 32-bit optional immediate. The first five fields are packed into one word (ten bits are left unused) while the immediate is held in a separate word.

produce different such updates. Each cycle, we *multiplex* over the twenty possible updates and only apply the current instruction's update. Multiplexers can be trivially implemented by large numbers of multiplications, but this is somewhat inefficient. Here, we show an improvement that uses vector-scalar multiplication to more efficiently implement a multiplexer. The key idea is that we organize values into a small table, then use vector-scalar multiplication to recursively discard half the table until only one table entry remains. The full lookup procedure uses $\log n$ OTs to compute a lookup on n elements.

Let T be a table of n elements $T_1, ..., T_{n/2}, T_{n/2+1}, ..., T_n$. Suppose $\mathcal{P}$ wishes to look up element i, and let i_1 be the first bit of this index: i_1 indicates if the needed element is in the first half or the second half of the table. To partially look up based on i_1, the parties compute the following expression:

$$(T_1, ..., T_{n/2}) + i_1 \cdot ((T_{n/2+1}, ..., T_n) - (T_1, ..., T_{n/2}))$$

If $i_1 = 0$, this expression computes $(T_1, ..., T_{n/2})$; otherwise, the expression computes $(T_{n/2+1}, ..., T_n)$. This expression requires a single OT of length $n/2$ secrets. We can recursively narrow down to a single element using $\log n$ OTs. The total length of OT secrets is n.

At the end of each cycle, we construct a table of three-tuples. Each tuple contains elements corresponding to one instruction: (1) the resultant program counter, (2) the resultant output-register value, and (3) the corresponding instruction op-code. After we perform a table lookup, we (1) update the program counter, (2) update the output register, and (3) check that the instruction's op-code matches the looked up op-code (see Section 7.4 for further discussion of this check).

7.4 Instruction Format and Decoding

On each cycle, the processor performs operations according to the current instruction. Instructions are stored in an efficient read-only memory that requires

only $2 \log N$ OTs per lookup [HK20a]. However, simply searching for the instruction is not sufficient. We must also ensure that the semantic action of the processor (how we update the program counter, registry, and main memory) is consistent with the instruction.

Figure 4 shows our instruction physical format. Each instruction holds an opcode, up to three register arguments, an immediate, and a flag that indicates if the immediate is used.

The first interesting aspect of our instruction format is that we store the immediate separately from the other fields. This makes reading the immediate trivial: no bit decomposition is needed to extract the immediate.

Our registry is implemented by BubbleCache. Hence, on each access, the RAM yields not only the looked up element, but also that element's index. Note also that the opcode is given as output from the cycle's multiplexer (see Section 7.3). Finally, $\mathcal{P}$ chooses $isImm$ as input during an OT that allows her to swap the third register argument for the immediate. Thus, each bit of the instruction's first word is represented twice: once in the physical instruction, and once in the corresponding parts of the processor cycle. Therefore, checking that the processor's action is consistent with the current instruction is extremely efficient: the parties use linear operations to compute a second copy of the instruction's first word, then check that the two copies are equal. This means that the first word of each instruction is never decomposed into bits, a significant improvement over [HK20a], where a large number of OTs were needed to decode each instruction.

7.5 Hardware Interrupts

Recall, our architecture jumps to program location `MEMFAULT` when the program accesses an illegal memory address (Section 6.5). Checking that a given memory address is legal requires performing comparisons and is expensive.

Fortunately, our processor supports skipped cycles, so we can handle interrupts cheaply: When an illegal access is attempted, $\mathcal{P}$ has no choice but to continually skip cycles. If she does not, BubbleCache will catch the attempt to index a non-existent address. With the processor successfully stuck, we need only make one change: our processor *periodically* checks if it is experiencing a fault (we check every $10,000$ cycles). This check is still relatively expensive, but is performed so infrequently that the cost is nearly irrelevant. If the check indicates a fault, our processor sets `pc` to `MEMFAULT`, and $\mathcal{P}$ can continue the proof from the new state.

Recall, our architecture maintains a protected region in the middle of memory. This region is implemented by simply deleting addresses from BubbleCache's address space.

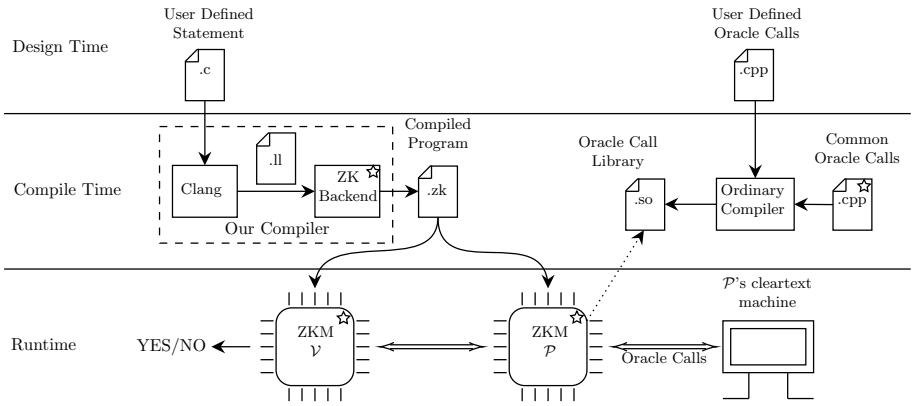

Figure 5: Our system design. Software components that we implemented are marked with a star; other software is either provided at design time or is off the shelf. To use our system, the programmer needs only provide a `C` program. The `C` program is translated to our instruction set by our custom compiler. The programmer provides the resulting assembly to both $\mathcal{V}$ and $\mathcal{P}$, who together run the program via the ZK protocol. When the protocol finishes, $\mathcal{V}$ either accepts or rejects the proof. As discussed in Section 6.4, $\mathcal{P}$'s runtime interfaces with her local system via prover oracle calls. We provide a library of common oracle calls, including implementations of low level operations and of system calls. The programmer may optionally augment the oracle call suite by compiling custom code and dynamically linking it with $\mathcal{P}$'s runtime.

8 ANSI C Implementation and System Design

The architecture described to this point requires an expert to craft the needed assembly programs and oracle calls. Developers should instead be empowered to write ZK proofs in a familiar way. We achieve this goal by implementing ANSI C95, a familiar and complete programming environment. Our implementation consists of two major components: (1) our C compiler lifts our ISA and allows programmers to write natural programs, and (2) our runtime library allows programmers to access data through the C standard library (libc) interface.

Our system is depicted in Figure 5. Users compose proofs as C programs that are compiled by our custom compiler, based on LLVM [LA04]. The resulting assembly runs on our ZKM. While running, the ZKM may communicate with $\mathcal{P}$'s clear-text system through oracle calls. Users may define and compile custom oracle calls to further extend this capability.

8.1 User Experience (UX)

We view the compiler and runtime libraries as the user-interface to our ZKM. Without a convenient interface, it is impractical for users to construct non-trivial ZK statements. We highlight the goals of our UX:

High-Level, Standard, Composable Programs Programmers should represent their logic in a standard high-level language, and that logic should be composable. Programmers should be able to use existing code to quickly and succinctly develop complex statements.

ZK execution that follows plaintext execution System and library calls, such as `fread` are transparently handled by our runtime library. When such calls are executed, honest $\mathcal{P}$'s runtime uses an oracle call that 'mirrors' the call on her local system. So for example, if the ZK program attempts to read a file, $\mathcal{P}$ generates the appropriate ZK input by reading that file on her local system. This mirroring makes ZK execution 'feel natural': performance aside, it is difficult to detect differences between running a program in clear-text and running the same program on our ZKM in prover mode.

Minimal instrumentation Standard C programs can be transformed into proof statements by simply adding the special instruction `QED`. Reaching `QED` constitutes a successful proof, so the programmer should place `QED` behind appropriate checks.

8.2 C Compiler

To support high-level, standard, composable programs, we provide a C compiler. Our compiler is built as a custom backend to the LLVM compiler toolchain, using the Clang C front-end. Clang translates C programs into LLVM's intermediate representation (IR), LLVM optimizes that IR, and a backend translates the IR

into the target architecture's instruction set. To support the ZKM, we add a new ZK backend that translates LLVM IR to our ISA. Much of this translation is well supported by our ISA, and while a significant effort, it is relatively straightforward. However, there are several classes of operations in the `C` language that need special handling.

First, there are division-like operations (divide, modulus, right-shift) which are not native to our ISA and must be emulated. We do so efficiently via oracle calls (cf Section 6.4). Our LLVM backend compiles division, right shift, and modulus operations into oracle calls with associated proofs.

For example, consider logical right-shift: $x \gg y$. Our compiler replaces right-shifts with appropriate oracle calls and associated proofs. Via the oracle call, $\mathcal{P}$ provides the right-shift result as input. Then, $\mathcal{V}$ checks in ZK that the value is correct (recall, $\mathcal{P}$ is untrusted). $\mathcal{V}$ checks two properties. First, $\mathcal{P}$'s low $32 - y$ bits must match the upper bits of x. Thus, the parties left-shift $\mathcal{P}$'s bits by y and compare the result with the upper bits of x. Second, $\mathcal{P}$'s high y bits must be zero. We use `AND` to zero out y's low bits, and we check that the result is zero. With these checks complete, $\mathcal{V}$ is confident that $\mathcal{P}$'s provided right shift result is correct.

Second, LLVM and `C` have byte-addressable memory, but our ISA only supports word-addressable memory. To allow sub-word granularity, our compiler translates sub-word memory accesses into full-word accesses. For loads, the compiler loads a word and masks out the needed bits. For stores, the compiler (1) loads the current value, (2) overwrites only the appropriate bits, and (3) performs a word-aligned store. With these operations, our compiler supports the full range of integer and pointer operations in the `C` language.

Third, there are floating point operations. Our compiler currently emulates floating point by translating each floating point operation into a library call. These library calls emulate floating point via integer arithmetic. While correct, this approach is slow and could be improved by future work.

Further improving our compiler, especially by incorporating optimizations, is an exciting direction for future work. We hope to add optimizations that take advantage of some of the ZK ISA's strengths (e.g. large register set), and hide some of its weaknesses (e.g. word-aligned memory). We also hope to explore architecture-compiler co-design, such as that of VLIW architectures [Fis83] to construct more efficient ZK systems.

8.3 Runtime System Support

Real-world programs interact with an external system: they communicate with other programs, read files, output data. However, our ISA only supports a primitive input interface. We therefore provide runtime support that lifts these low-level operations into a `C` standard library interface (`libc`). Our `libc` enables the ZKM to interact with processes and files on $\mathcal{P}$'s host system, allowing standard `C` programs, such as `sed` and `gzip`, to run on the ZKM without modification.

We base our `libc` implementation on Newlib [RHS20]. Newlib builds the higher-level portions of `libc` on top of `libgloss`, a collection of the standard library's lowest level behaviors. We provide a custom `libgloss` implementation, translating its sixteen system interaction functions, such as `_read` (the `libgloss` version of `read`), into oracle calls. Figure 6 lists a simplified version of our `_read` implementation, which provides an example of how our entire standard library is structured.. With `libgloss` implemented, Newlib provides the remaining standard library functionality, and thus we fully support `libc`.

8.4 System Limitations

Our system supports ANSI `C95` programs. However, many applications rely on POSIX interfaces and/or features of newer `C` standards (`C11`, `C17`) beyond ANSI `C95`. In many cases, integrating these more complex interfaces, while possible, would require significant additional engineering. In other cases, additional 'hardware' support would be required. For instance, `mprotect` requires memory protection, requiring our ZKM to add an MPU.

While the security of our ZKM is formally proven, the correctness of our *compiler* is not. Consequently, a skeptical $\mathcal{P}$ or $\mathcal{V}$ should inspect the compiled assembly. One possible future direction is to formally verify portions of the compiler [Ler09].

Our `libc` implementation does not validate data provided by the external system: such input is seen as part of $\mathcal{P}$'s witness. For instance, a database application may assume it always reads from a well-formed database. Our `libc` will only return data read from the file, but will not check its contents. If an application has input constraints, it should include logic that checks those constraints. Put another way, only the application itself is authenticated in ZK. The *execution environment* is fully under $\mathcal{P}$'s control, and she can arbitrarily and adaptively alter it: e.g., she can change files, delete files, or start/stop interacting processes. We leave embedding more of the execution environment in the ZK proof as future work.

ORAM size is an important consideration: BubbleCache is stored in corresponding data structures on both $\mathcal{P}$ and $\mathcal{V}$'s systems, and so the size-n BubbleCache is bounded by the hardware RAM of their systems. Additionally, the largest permutations employed by BubbleCache impose significant overhead: the largest permutation requires $\mathcal{P}$ and $\mathcal{V}$ to execute $\frac{1}{2}n \log n$ OTs. Further, we perform all OTs at the start of the protocol. This can be taxing for weak $\mathcal{P}/\mathcal{V}$.

Natural directions for future work include more efficient floating point support, compiler optimizations, and adding a linker, among others.

9 Evaluation

In this section, we describe the implementation and evaluation of our system. We emphasize our system's concrete performance: our system runs at up to

11.89KHz with a substantial RAM (see Figure 10 for detailed performance reports, including clock rate, on several benchmarks). This is a $\approx 5.5\times$ improvement over the clock-rate achieved by [HK20a] for a more expressive instruction set.

9.1 Implementation Details and Testing Environment

We implemented our approach in `C/C++`. Our front-end compiler and standard library together are $\approx$ 6KLOC. Our backend cryptographic ZK machine implementation is $\approx$ 3KLOC. Our compiler is based on `LLVM version 10.0.0` and `newlib 3.3.0`. We instantiated Oblivious Transfer using the recent Ferret correlated OT [YWL+20] as implemented by the EMP Toolkit [WMK16]. All benchmarks were run on a commodity laptop: a MacBook Pro with an Intel Dual-Core i5 3.1 GHz processor and 8GB of RAM. The two parties communicated over a simulated 1Gbps LAN with 2ms latency. Unless otherwise stated, we instantiate BubbleCache using delay parameter $C = \frac{1}{2}$.

9.2 Bug Analysis and Detection Methods

In our first experiment, we demonstrate our system's ability to handle real-world programs. We pulled buggy versions of two popular Linux programs, `sed` and `gzip`, off the shelf. We added minimal instrumentation by inserting `QED` calls at appropriate locations. Our system successfully proved in ZK the existence of the bug in each program.

9.2.1 `sed`

Older versions of `sed` (we use 1.17) contain a segmentation fault bug listed in the Software-artifact Infrastructure Repository (SIR) [DER05]: omitting the final newline character from the input file results in a length underflow, which causes `sed` to attempt to overwrite its entire address space. This version of `sed` has 5.4KLOC. The bug exercises undefined behavior and could exhibit itself in a number of ways. Our architecture detects this bug through our memory protection mechanism. We configure our memory hardware interrupt, `MEMFAULT`, to hold `QED` (see Section 6.5), causing a fault to conclude the proof. The hardware interrupt is triggered when `sed` attempts to write to the protected region of memory between the stack and heap.

We instrumented BubbleCache with 2^{13} words of memory. `QED` is reached after $390,002$ cycles, shortly after completion of a hardware interrupt. The proof ran in 36.1s, yielding a clock-rate of 11.1KHz. The execution incurred $6,470$ cache misses and $45,912$ skipped cycles.

9.2.2 `gzip`

The bug (CVE-2005-1228 [CVE05]) allows an attacker to illegally write to an arbitrary directory. When `gzip` decompresses a zip file, the output directory

is intended to be named according to a prefix of the input file name. Under certain inputs, `gzip` will erroneously write to an *arbitrary* directory chosen by the attacker. We used `gzip 1.3`, which is believed to be the last version with this bug. This version of `gzip` has 5.4KLOC. We detect the bug by placing string comparison logic immediately before opening an output file.

We used BubbleCache with 2^{17} words of memory and $C = \frac{1}{2}$. The proof uses $44,092$ cycles and runs in 6.5s, yielding a clock-rate of 6.8KHz. `gzip` runs slower than `sed` because we use a larger RAM and do not issue enough instructions to fully amortize the largest BubbleCache partitions. The execution incurred 554 cache misses and $2,868$ skipped cycles.

9.3 BubbleCache Performance

9.3.1 Comparison to BubbleRAM

We compare the communication cost of BubbleCache to that of BubbleRAM. We ran the trivial program, a single `QED` instruction, but ensured that the proof ran for sufficient cycles to fully amortize RAM cost. RAM cost is program independent: the processor permutes and accesses RAM on every cycle, regardless of the program. We instantiated RAM with both BubbleCache and BubbleRAM of various sizes. Figure 7 plots the resulting RAM communication consumption. The results clearly show BubbleCache's asymptotic advantage. With 2^{17} memory words, BubbleCache improves communication by more than $8\times$ over BubbleRAM.

9.3.2 Cache Miss Rate and Impact

BubbleCache introduces the possibility of a cache miss. To better understand BubbleCache performance, we ran our system with different configurations of RAM and against different benchmarks. A full tabulation of the results is in Appendix E. Figure 8 gives a high level summary of the results: in our benchmark suite, despite the fact that RAM is only a single component of our processor step, using BubbleCache provides substantial improvement.

9.4 Prover Oracle Call Shortcuts

As discussed in Section 6.4, oracle calls can shortcut the evaluation of functions that are expensive to compute but cheap to verify. We demonstrate this with a concrete example. We implemented both merge sort and 'input and prove' sort. In the latter, $\mathcal{P}$ locally sorts the array, then provides the sorting permutation as input. In ZK, the players check that the input is indeed a permutation, apply the permutation, and check the list is sorted. This algorithm uses only $\Theta(n)$ ZK computation.

We ran the two algorithms on lists of 10, 100, and 1000 elements. 'Input and prove' sort used respectively $1.47\times$, $3.92\times$, and $6.35\times$ fewer ZK instructions than merge sort.

9.5 Maximum Memory Size

In our benchmarks reported in Figure 10, we configured RAM with the minimum size needed to correctly execute each program. We did so to demonstrate maximum performance. However, our system can also be configured with much larger RAM. Thus, we experimented with the maximum size of RAM that our system can handle. We ran our `gzip` experiment with 2^{24} words, or 32MB, of main memory. The proof completed successfully in 199s. The proof is substantially slower than when run with 2^{17} words because of large partitions performed on the first RAM access. When we attempted 2^{25} words, our modest laptop exhausted its memory resource and the program crashed. With programming improvements or by using better hardware, we could scale to larger main memory.

Acknowledgments

We thank our S&P'21 reviewers for their many insightful comments.

This work was supported in part by NSF award #1909769, by a Facebook research award, by Georgia Tech's IISP cybersecurity seed funding (CSF) award, and by Sandia National Laboratories, a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525. This material is also based upon work supported in part by DARPA under Contract No. HR001120C0087. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2087–2104. ACM Press, October / November 2017.

[ALSZ13] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 535–548. ACM Press, November 2013.

[BBB+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334. IEEE Computer Society Press, May 2018.

[BBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018. https://eprint.iacr.org/2018/046.

[BBHR19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 701–732. Springer, Heidelberg, August 2019.

[BCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 90–108. Springer, Heidelberg, August 2013.

[BCG+19] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 489–518. Springer, Heidelberg, August 2019.

[BCR+19] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for R1CS. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 103–128. Springer, Heidelberg, May 2019.

[BCTV14a] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 276–294. Springer, Heidelberg, August 2014.

[BCTV14b] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014*, pages 781–796. USENIX Association, August 2014.

[Bel66] L. Belady. A study of replacement algorithms for virtual-storage computer. *IBM Syst. J.*, 5:78–101, 1966.

[BFH+20] Rishabh Bhadauria, Zhiyong Fang, Carmit Hazay, Muthuramakrishnan Venkitasubramaniam, Tiancheng Xie, and Yupeng Zhang. Ligero++: A new optimized sublinear IOP. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 20*, pages 2025–2038. ACM Press, November 2020.

[CDG+17] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1825–1842. ACM Press, October / November 2017.

[CFH+15] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *2015 IEEE Symposium on Security and Privacy*, pages 253–270. IEEE Computer Society Press, May 2015.

[CVE05] Common Vulnerabilities and Exposures. CVE-2005-1228. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-1228`, 2005.

[DER05] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[Fis83] Joseph A. Fisher. Very long instruction word architectures and the eli-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, ISCA '83, page 140–150, New York, NY, USA, 1983. Association for Computing Machinery.

[FNO15] Tore Kasper Frederiksen, Jesper Buus Nielsen, and Claudio Orlandi. Privacy-free garbled circuits with applications to efficient zero-knowledge. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 191–219. Springer, Heidelberg, April 2015.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 626–645. Springer, Heidelberg, May 2013.

[GMO16] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster zero-knowledge for Boolean circuits. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 1069–1083. USENIX Association, August 2016.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *17th ACM STOC*, pages 291–304. ACM Press, May 1985.

[GMW91] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. *J. ACM*, 38(3):690–728, July 1991.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Heidelberg, May 2016.

[HK20a] David Heath and Vladimir Kolesnikov. A 2.1 KHz zero-knowledge processor with BubbleRAM. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 20*, pages 2055–2074. ACM Press, November 2020.

[HK20b] David Heath and Vladimir Kolesnikov. Stacked garbling for disjunctive zero-knowledge proofs. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part III*, volume 12107 of *LNCS*, pages 569–598. Springer, Heidelberg, May 2020.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th ACM STOC*, pages 21–30. ACM Press, June 2007.

[JKO13] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 955–966. ACM Press, November 2013.

[KK13] Vladimir Kolesnikov and Ranjit Kumaresan. Improved OT extension for transferring short secrets. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 54–70. Springer, Heidelberg, August 2013.

[KKW18] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 525–537. ACM Press, October 2018.

[LA04] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[LO13] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 719–734. Springer, Heidelberg, May 2013.

[PHGR13] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252. IEEE Computer Society Press, May 2013.

[RHS20] Red Hat Software. Newlib. `https://sourceware.org/newlib/`, January 2020.

[SAGL18] Srinath Setty, Sebastian Angel, Trinabh Gupta, and Jonathan Lee. Proving the correct execution of concurrent services in zero-knowledge. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 339–356, Carlsbad, CA, October 2018. USENIX Association.

[SHS+15] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symposium on Security and Privacy*, pages 411–428. IEEE Computer Society Press, May 2015.

[Wak68] Abraham Waksman. A permutation network. *J. ACM*, 15(1):159–163, January 1968.

[WGMK16] Xiao Shaun Wang, S. Dov Gordon, Allen McIntosh, and Jonathan Katz. Secure computation of MIPS machine code. In Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows, editors, *ESORICS 2016, Part II*, volume 9879 of *LNCS*, pages 99–117. Springer, Heidelberg, September 2016.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. `https://github.com/emp-toolkit`, 2016.

[WSR+15] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS 2015*. The Internet Society, February 2015.

[WYKW20] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. Cryptology ePrint Archive, Report 2020/925, 2020. `https://eprint.iacr.org/2020/925`.

[XZZ+19] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 733–764. Springer, Heidelberg, August 2019.

[YWL+20] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 20*, pages 1607–1626. ACM Press, November 2020.

A Succinct and Non-Interactive ZK Related Work

Ishai et al. introduced the 'MPC-in-the-head' technique [IKOS07]: here, $\mathcal{P}$ emulates in her head an MPC evaluation of the proof among virtual players. $\mathcal{V}$ checks random portions of the evaluation transcript and thus gains confidence that the prover has a witness. By allowing $\mathcal{V}$ to inspect transcripts of only some virtual players, the protocol protects $\mathcal{P}$'s secret. [IKOS07] inspired a number of subsequent MPC-in-the-head works, e.g. [GMO16, CDG+17, AHIV17, KKW18].

Succinct non-interactive arguments of knowledge (SNARK) are extremely small proofs that can be quickly verified [GGPR13, PHGR13, BCG+13, CFH+15, Gro16]. Early SNARKs require a semi-trusted party, so more recent works developed STARKs (succinct transparent arguments of knowledge) [BBHR18]. STARKs do not require trusted setup and rely on more efficient primitives. [BFH+20]'s Ligero++ combines techniques of [AHIV17, BCR+19].

[HK20b] extensively analyzed many of the above works: namely [KKW18], Ligero, Aurora, Bulletproofs, STARK, and Libra [KKW18, AHIV17, BCR+19, BBB+18, BBHR19, XZZ+19]. Their analysis demonstrates that while these works have excellent performance in certain aspects (e.g., small proof size, fast verification time, non-interactivity), they struggle to handle large proofs motivated by problems like proving the existence of a bug in a program.

B [HK20a] Protocol, Extended

Section 5 introduced [HK20a]'s arithmetic ZK protocol and its authenticated sharing representation. Here, we continue by showing how the protocol primitives are implemented.

Proofs of zero and of using correct inputs $\mathcal{P}$ must frequently prove that her intermediate inputs are chosen correctly (cf, e.g., Section 6.1.4). This is achieved by proving equality of corresponding values, which, in turn is achieved by proving that particular shares each encode zero. This is simple: $\mathcal{P}$ sends her share to $\mathcal{V}$ and $\mathcal{V}$ checks that the two shares sum to zero. We use a simple

optimization: rather than sending each proof of zero separately, $\mathcal{P}$ instead accumulates a hash digest of all proofs and sends this to $\mathcal{V}$. Thus, $\mathcal{P}$ needs only send κ bits to $\mathcal{V}$ to prove the validity of every zero check.

Linear Operations Sharings support efficient linear operations. Linear operations do not require communication; the parties need only operate locally on their respective shares.

- To add sharings, parties locally add the shares:

$$\begin{aligned}\llbracket x \rrbracket + \llbracket y \rrbracket &= \langle X, x\Delta - X\rangle + \langle Y, y\Delta - Y\rangle \\ &\triangleq \langle X+Y, (x+y)\Delta - (X+Y)\rangle = \llbracket x+y \rrbracket\end{aligned}$$

  To subtract sharings, parties similarly subtract the shares.

- To multiply a sharing by a public constant, the parties locally multiply the shares by the constant:

$$c\llbracket x \rrbracket = c\langle X, x\Delta - X\rangle \triangleq \langle cX, cx\Delta - cX\rangle = \llbracket cx \rrbracket$$

- Parties encode constants as follows: $\langle c\Delta, 0\rangle = \llbracket c \rrbracket$

Vector-Scalar Multiplication [HK20a]'s protocol provides non-linear operations via a special form of share multiplication where one share is known to be in $\{0,1\}$. We give a simple generalization where instead $\mathcal{P}$ scales an entire vector by a private bit. This generalization can achieve [HK20a]'s multiplication by incorporating public constants and a zero proof; in fact, the composed protocol is *identical* to [HK20a]'s. Our extension also allows other usage, such as fast small table lookup (Section 7.3). We note that [HK20a] did mention some special case usage of vector-scalar multiplication, but did not frame the formalization as generally as the following:

Let $x \in \{0,1\}$ be held by $\mathcal{P}$ and let $y_1, ..., y_n \in \mathbb{Z}_p$ be a vector. The parties hold sharings $\llbracket y_1 \rrbracket, ..., \llbracket y_n \rrbracket$ and wish to compute $\llbracket xy_1 \rrbracket, ..., \llbracket xy_n \rrbracket$ (while $\mathcal{P}$'s input x is not authenticated, it could be verified later by an appropriately applied proof of zero). First, $\mathcal{P}$ locally multiplies her shares by x. Thus the parties together hold:

$$\langle Y_1, xy_1\Delta - xY_1\rangle, ..., \langle Y_n, xy_n\Delta - xY_n\rangle$$

These intermediate sharings are invalid: the shares in the ith sharing do not sum to $y_i\Delta$. To resolve this, the parties participate in a single 1-out-of-2 OT where $\mathcal{V}$ acts as the sender. $\mathcal{V}$ uniformly draws n values $Y_i' \in_\$ \mathbb{Z}_p$ and allows $\mathcal{P}$ to choose between the following two vectors:

$$Y_1', ..., Y_n' \qquad Y_1' - Y_1, ..., Y_n' - Y_n$$

$\mathcal{P}$ chooses based on x and hence receives as output the vector $Y_1' - xY_1, ..., Y_n' - xY_n$. The parties can now compute a valid sharing for each vector index:

$$\langle Y_i', xy_i\Delta - xY_i - (Y_i' - xY_i)\rangle = \langle Y_i', xy_i\Delta - Y_i'\rangle = \llbracket xy_i \rrbracket$$

A vector-scalar multiplication of a length n vector requires a 1-out-of-2 OT of $n\lceil \log p \rceil$-bit secrets. Since the transmitted terms Y_i' are random, it suffices to use *correlated* OT [ALSZ13, KK13]. In practice, we instantiate multiplication with the Ferret correlated OT technique [YWL+20].

Security. We stress that there is no security consequence in using the above generalization of [HK20a]'s multiplication primitive: [HK20a]'s correctness, soundness, and verifiability proofs of the underlying garbling scheme are each trivially adjusted to accommodate this more powerful primitive. Thus, the extended share algebra of [HK20a] is a malicious-verifier ZKP system.

C Arithmetic Logic Unit

Figure 9 lists the core of the ALU. Our ALU *aggressively* amortizes multiplications: in total, our ALU uses only 65 OTs. The listed code does not handle `MOV` (which is trivial) or `CMOV` (which is handled by one additional OT). The ALU outputs are multiplexed by a small table lookup (see Section 7.3).

`ADD`, `SUB`, and `MUL` can each *overflow* $\mathbb{Z}_{2^{32}}$. The highest overflow comes from our multiplication algorithm, which can produce values as high as $32 \cdot (2^{32} - 1)$. We account for overflow by allowing $\mathcal{P}$ to input a value $x \in \{0..31\}$. The parties subtract $2^{32} \cdot [\![x]\!]$ from the ALU output, allowing $\mathcal{P}$ to subtract off the top bits. At first, this may seem insecure. However, note that the ALU bit decomposes each input register on each cycle, and the validity of these decompositions are checked. These checks can succeed only if the registers hold valid $\mathbb{Z}_{2^{32}}$ elements. Thus, it is possible for a malicious $\mathcal{P}$ to store an element not in $\mathbb{Z}_{2^{32}}$ in the registry, but she cannot *operate* on such an illegal element without being caught.

D Proofs

Section 7 claimed Theorem 1: our microarchitecture is sound and complete. We now formalize in detail and prove this claim. Let κ be the computational and σ be the statistical security parameters (e.g., $\kappa = 128, \sigma = 40$).

To formalize security, we must capture *all* of $\mathcal{P}$'s OT choices. Therefore, we define the notion of an *extended witness*, which includes not only the program input, but also all 'supporting' OT choices needed to complete the proof (e.g., during bit decomposition $\mathcal{P}$ selects OT outputs corresponding to the decomposed integer).

Definition 1 (Extended Witness)**.** *The* extended witness *is the complete collection of $\mathcal{P}$'s inputs, including supporting choices that are not syntactically part of her witness.*

Definition 2 (Proof Microarchitecture)**.** *A proof microarchitecture μ is a protocol $(\mathcal{P}, \mathcal{V})$. Let P be a program in our ISA. $\mathcal{P}$ takes as input P and an extended witness I. $\mathcal{V}$ takes as input P. At the end of the protocol, $\mathcal{V}$ outputs* 0 *or* 1*.*

Definition 3 (Proof Microarchitecture Completeness)**.** *A proof microarchitecture μ* completely *implements our architecture if for all programs P in our ISA and all extended witnesses I such that $P(I)$ reaches* `QED`*, $\mathcal{P}(P, I)$ causes $\mathcal{V}$ to output* 1*.*

Theorem 3. *If the prime field modulus p is greater than 2^{37}, then our microarchitecture (Construction 2) is a* complete *implementation of our architecture (Construction 1) in the OT-hybrid model.*

Proof. By inspection of the microarchitecture (Section 7) including the correctness of BubbleCache (Section 7.1), of [HK20a]'s oblivious ROM, of small table lookup (Section 7.3), and of our ALU (Figure 9). Our microarchitecture is built on [HK20a]'s share algebra, which, given that OT is correct, is complete. At a high level, our microarchitecture is a relatively straightforward implementation of the architecture. We mention several of the interesting points.

- BubbleCache introduces *cache misses*, which are not part of our architecture. We nevertheless correctly implement the architecture because (1) $\mathcal{P}$ skips cycles that would otherwise experience a cache miss and (2) because BubbleCache makes progress towards a cache hit during skipped cycles: our processor cannot become stuck.
- The architecture includes *hardware interrupts* that are issued on illegal memory accesses. Our microarchitecture does not *immediately* issue hardware interrupts. However, $\mathcal{P}$ skips cycles in order to simply wait until we *periodically* check for an illegal memory access. Because we periodically check, we always make progress towards an interrupt if the processor is in a fault state.
- Our architecture operates over values in $\mathbb{Z}_{2^{32}}$, but our microarchitecture operates over values in $\mathbb{Z}_p$. However, we compute mod 2^{32} on every cycle, and thus while our representation can capture any value in $\mathbb{Z}_p$, our registry and main memory will never store a value $\geq 2^{32}$. Additionally, no operation will overflow the field: multiplication has the largest outputs, with a maximum output value of $32 \cdot (2^{32} - 1) < 2^{37}$.
- Our architecture performs only one instruction per cycle, but our microarchitecture must emulate *every* instruction on *every* cycle. This difference is easily accounted for by (1) issuing dummy memory look-ups when a register/address is not needed, (2) having $\mathcal{P}$ provide a dummy input on cycles when input is not needed, and (3) multiplexing (via small table lookup) the effect of all instructions based on the current instruction opcode. Thus, while we *compute* the effect of each instruction, we only *apply* the effect of the active instruction.

Our microarchitecture is complete. □

Even though our microarchitecture is described in the OT-hybrid model, we still need computational assumptions: we perform zero-tests by comparing hashes of a set of zero labels.

To define soundness, we define a syntactic interface to the prover adversary $\mathcal{A}$. $\mathcal{A}$ takes the program P, the extended witness I, and her *share* of I as input: $\mathcal{A}(P, I, [\![I]\!])$. $\mathcal{A}$'s interface is different from $\mathcal{P}$'s: $\mathcal{A}$ takes as input the already-encoded shares $[\![I]\!]$. That is, $\mathcal{A}$ does not participate in the OT step of μ, and instead directly receives $\mathcal{P}$'s shares of I.

Definition 4 (Proof Microarchitecture Soundness)**.** *A proof microarchitecture* μ soundly *implements our architecture if for all programs* P *in our ISA, all extended witnesses* I *such that our architecture does not reach* `QED` *within* $poly(1^\kappa)$ *cycles, and all probabilistic poly-time adversaries* $\mathcal{A}$*, the probability that* $\mathcal{A}(P, I, [\![I]\!])$ *causes* $\mathcal{V}$ *to output* 1 *is negligible in* κ*.*

Similar to [JKO13]'s definition of soundness, the above ensures that malicious $\mathcal{P}$ cannot win unless she has input labels corresponding to I such that $P(I)$ terminates in `QED`.

Theorem 4. *Assuming a collision resistant hash function, that the prime field modulus* $p > 2^{37}$*, and that* $\lfloor \log p \rfloor \geq \sigma$*, our microarchitecture (Construction 2) is a* sound *(w.r.t.* σ*) implementation of our architecture (Construction 1) in the OT-hybrid model.*

Proof. By soundness of [HK20a]'s share algebra extended with our generalized vector-scalar multiplication protocol. [HK20a]'s share algebra can be framed as a privacy-free *garbling scheme*, and hence is sound via [JKO13]'s protocol.

Our microarchitecture is built on share algebra, so primitive operations are sound. We introduce opportunities to cheat by allowing $\mathcal{P}$ to freely choose certain inputs, but we account for each of these opportunities by forcing $\mathcal{P}$ to prove certain values are equal to zero (as an optimization, we use a collision-resistant hash function to compute and verify only a digest of *all* zero proofs):

- To compute mod 2^{32}, $\mathcal{P}$ freely subtracts the top bits from the ALU output. Suppose $\mathcal{P}$ cheats and subtracts top bits that do not correctly compute mod 2^{32}. The ALU output is subsequently stored in the registry. The next time the invalid register is accessed, it will be decomposed into 32 bits, and the validity of the decomposition will be checked. Since the register value does not fit in 32 bits, this check will fail and $\mathcal{V}$ will reject.

- Each cycle, the processor updates `pc` and the registry by multiplexing over the effects of all possible instructions. To multiplex these effects, $\mathcal{P}$ freely chooses a row of a small lookup table. Suppose $\mathcal{P}$ cheats and chooses a row that does not correspond to the current instruction. One of the columns of the table is the instruction opcode. Before the cycle terminates, $\mathcal{P}$ must prove that the looked up opcode (together with the accessed registry indices) matches the current instruction. Since $\mathcal{P}$ cheated and does not have the correct opcode, she cannot pass this check.

$\mathcal{P}$ cannot cheat by skipping cycles. When a cycle is skipped, the processor makes no progress, except to partition the content of BubbleCache.

$\mathcal{P}$ *can* cheat by guessing a share that does not arise from the program execution. Most directly, she can guess a `pc` share that moves the processor to `QED`. As discussed in Section 5, this guess succeeds with probability $\frac{1}{p-1}$, and hence only succeeds with probability negligible in σ. Even if there are multiple `QED` instructions, $\mathcal{P}$'s chances are not improved: she not only has to reach `QED`, but also needs to know which specific `pc` value she reached such that she can appropriately prove the current `pc` value is in the ROM. This is a particular case of a more general property of the protocol: to cheat, $\mathcal{P}$ must both know a valid share *and* know the corresponding clear-text value.

Our microarchitecture is sound. □

Theorem 5. *Our microarchitecture is malicious-verifier Zero Knowledge.*

Proof. Immediate. The [HK20a] protocol is malicious-verifier Zero Knowledge, as proved by [JKO13]. □

Theorems 3 to 5 together imply Theorem 1.

E Cache Miss Rate, Extended

The Figure 8 statistics are extended in Figure 10.

```
#define READ_ORACLE 1
#define FD_REGISTER ...
#define LEN_REGISTER ...
...
int _read(int fd, char *buf, int len) {
  proveroracle(READ_ORACLE, fd, len);
  int rc = readtape();
  if (rc > len) { HALT; }
  for (int i = 0; i < rc; i++) {
    buf[i] = readtape() & 0xFF;
  }
  return rc;
}
...
void read_orc(state* s) {
  int fd = s->registry[FD_REGISTER];
  int len = s->registry[LEN_REGISTER];
  char data[] = new char[len];
  // calls Linux's read function
  int rc = read(fd, data, len);
  writetape(s->input, rc);
  for (int i = 0; i < rc; i++) {
    writetape(s->input, data[i]);
  }
  delete[] data;
}
REGISTER_ORACLE(READ_ORACLE, &read_orc);
```

Figure 6: The ZK procedure for a file-system read (top) and the corresponding oracle call procedure run only by $\mathcal{P}$ (bottom). `_read` is run explicitly by the ZK machine. When $\mathcal{P}$ executes this procedure and reaches the `proveroracle` call, she temporarily escapes the ZK processor, runs the procedure `read_orc` on her local machine to populate the machine's input tape, and then re-enters the ZK processor. In contrast, $\mathcal{V}$ performs a no-op at the `proveroracle` call. Importantly, `_read` checks that $\mathcal{P}$ provided a valid number of bytes. `REGISTER_ORACLE` is a helper procedure that lets the programmer extend $\mathcal{P}$'s oracle call table. Note, because oracle calls (e.g., `read_orc`) are run in cleartext on $\mathcal{P}$'s machine, they are compiled separately from the proof statements and can be written in `C++`.

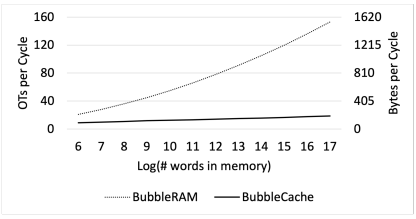

Figure 7: BubbleCache's and BubbleRAM's amortized communication consumption per cycle as a function of the size of RAM. Each OT swaps two pairs of words and sends 81 bits (40 per pair element and 1 overhead bit).

Benchmark	**RAM Size (Words)**	**Time**	**Comm.**	**Clock rate**
Sort (500)	2^{15}	1.78×	1.44×	1.92×
Sum (5,000)	2^{15}	1.34×	1.54×	1.36×
`sed`	2^{13}	1.26×	1.29×	1.40×
`gzip`	2^{17}	1.15×	1.31×	1.23×

Figure 8: Overall system performance improvement when implementing RAM with BubbleCache versus with BubbleRAM. We tabulate (1) wall-clock time reduction, (2) total communication (including non-RAM actions) reduction, and (3) clock-rate (i.e., Hz) increase. We configured ZKM, setting RAM to both BubbleCache and BubbleRAM for each of four benchmarks: calling merge sort on 500 random numbers, summing a list of $5,000$ random numbers, `gzip`, and `sed` bug proofs. An expanded table with additional metrics is given in Figure 10.

$\mathsf{ALU}([\![x]\!], [\![y]\!])$:

$[\![z]\!] \leftarrow [\![1]\!]$ ▷ z will in the end denote $x = 0$

for $i \in 0..31$:

$([\![1 - x_i]\!], [\![z]\!]) \leftarrow (1 - x_i) \cdot ([\![1]\!], [\![z]\!])$ ▷ Decompose x into bits and check if x is zero.

$[\![pow]\!] \leftarrow [\![1]\!]$ ▷ pow will in the end denote $2^y \bmod 2^{32}$

for $i \in 0..31$:

▷ Decompose y into bits, compute bit operations, multiply (with overflow), and compute 2^y

$$[\![2^i x \bmod 2^{32}]\!] = \sum_{j=0}^{31-i} 2^{i+j} \cdot [\![x_j]\!]$$

$[\![\delta_{pow}]\!] \leftarrow ((2^{(2^i)} \bmod 2^{32}) - 1) \cdot [\![pow]\!]$

$([\![y_i]\!], [\![(x \wedge y)_i]\!], [\![y_i \cdot (2^i x \bmod 2^{32})]\!], [\![\delta_{pow}]\!]) \leftarrow y_i \cdot ([\![1]\!], [\![x_i]\!], [\![2^i x \bmod 2^{32}]\!], [\![\delta_{pow}]\!])$

$[\![(x \vee y)_i]\!] \leftarrow [\![x_i]\!] + [\![y_i]\!] - [\![(x \wedge y)_i]\!]$

$[\![(x \oplus y)_i]\!] \leftarrow [\![(x \vee y)_i]\!] - [\![(x \wedge y)_i]\!]$

$[\![pow]\!] \leftarrow [\![pow]\!] + [\![\delta_{pow}]\!]$

$$\mathsf{prove}\ [\![x]\!] - \left(\sum_{i=0}^{31} 2^i \cdot [\![x_i]\!]\right) = [\![0]\!] \quad ; \quad \mathsf{prove}\ [\![y]\!] - \left(\sum_{i=0}^{31} 2^i \cdot [\![y_i]\!]\right) = [\![0]\!]$$

$$\mathsf{return}\ \left([\![x + y]\!], [\![2^{32} + x - y]\!], \left(\sum_{i=0}^{31} [\![y_i \cdot (2^i x \bmod 2^{32})]\!]\right), [\![x \oplus y]\!], [\![x \wedge y]\!], [\![x \vee y]\!], [\![z]\!], [\![y_{31}]\!], [\![pow]\!]\right)$$

Figure 9: The core of our ALU computes `ADD`, `SUB`, `MUL`, `XOR`, `AND`, `OR`, `EQZ`, `MSB`, and `POW2`. We denote the ith bit of a value x by writing x_i. If a variable appears outside a share and is not a constant, it indicates that this is a choice made by honest $\mathcal{P}$. The full algorithm consumes only 64 OTs. Addition, subtraction, and multiplication all include the possibility of an overflow, which is dealt with elsewhere in the processor.

Benchmark	RAM	Instrs.	# Access	Cycles	Time (s)	Cache Misses	IPC	Comm. (MB)	Millions of OTs	Clock (KHz)
Sort (500)	BubbleRAM	524399	135611	524399	99.0	0.00%	1.000	1755	144.0	5.30
	BubbleCache 1/6			524399	65.9	0.00%	1.000	1409	108.0	7.96
	BubbleCache 1/3			527053	53.9	0.49%	0.995	1212	87.4	9.78
	BubbleCache 1/2			565503	55.6	5.57%	0.927	1216	85.0	10.17
	BubbleCache 2/3			593171	67.3	5.92%	0.884	1249	86.5	8.81
	BubbleCache 5/6			809246	86.6	20.43%	0.648	1674	115.0	9.34
	BubbleCache 1			828633	87.3	20.52%	0.633	1708	117.0	9.50
Sum (5000)	BubbleRAM	280025	76377	280025	34.4	0.00%	1.000	939	77.0	8.15
	BubbleCache 1/6			280025	28.5	0.00%	1.000	754	57.8	9.83
	BubbleCache 1/3			280039	25.2	0.01%	1.000	647	46.7	11.11
	BubbleCache 1/2			283077	25.6	2.78%	0.989	611	42.8	11.05
	BubbleCache 2/3			283333	24.4	2.79%	0.988	598	41.4	11.63
	BubbleCache 5/6			381466	32.1	16.68%	0.734	790	54.3	11.89
	BubbleCache 1			382554	32.6	16.70%	0.732	788	54.0	11.74
`sed`	BubbleRAM	344086	56525	350002	45.4	0.00%	0.983	1130	89.7	7.71
	BubbleCache 1/6			350002	37.6	0.00%	0.983	973	73.4	9.30
	BubbleCache 1/3			350002	35.1	1.02%	0.983	854	61.1	9.98
	BubbleCache 1/2			390002	36.1	11.45%	0.882	874	60.8	10.81
	BubbleCache 2/3			400002	37.1	11.91%	0.860	943	65.1	10.79
	BubbleCache 5/6			530002	53.1	27.94%	0.649	1213	83.2	9.98
	BubbleCache 1			540002	54.9	28.10%	0.637	1224	83.8	9.83
`gzip`	BubbleRAM	41224	6687	41224	7.5	0.00%	1.000	210	17.1	5.53
	BubbleCache 1/6			41224	6.9	0.00%	1.000	177	13.7	6.01
	BubbleCache 1/3			41458	6.3	0.76%	0.994	162	12.2	6.53
	BubbleCache 1/2			44092	6.5	8.28%	0.935	161	11.9	6.82
	BubbleCache 2/3			45628	6.5	8.78%	0.903	162	11.9	7.02
	BubbleCache 5/6			60643	7.3	29.85%	0.680	186	13.5	8.28
	BubbleCache 1			62499	7.7	29.98%	0.660	188	13.6	8.15

Figure 10: Performance characteristics of our system with various RAM configurations and on various benchmarks. We record (1) the total number of instructions needed to reach `QED`, (2) the number of `LOAD`/`STORE` instructions in the execution, (3) total cycles (including skipped cycles), (4) the total execution time, (5) the cache miss rate (i.e., the ratio of missed loads/stores to `LOAD`/`STORE` instructions), (6) the number of instructions executed per cycle (IPC), (7) the total communication in MB (8) the number of OTs in millions, and (9) our system's clock-rate in KHz. Note that each cache miss can be responsible for more than one skipped cycle. Hence, cache miss rate and IPC can differ. For example, the `sed` bug causes one cache miss for BubbleRAM on the illegal access, but that single miss incurs almost $6,000$ skipped cycles while waiting for a hardware interrupt. We ran each of four benchmarks: merge sort on a list of 500 random numbers, summing a list of $5,000$ numbers, our `sed` bug benchmark, and our `gzip` bug benchmark. We ran each benchmark using BubbleRAM and using BubbleCache configured with various delay constants C.