A 2.1 KHz Zero-Knowledge Processor with BubbleRAM

David Heath
Georgia Tech
heath.davidanthony@gatech.edu

Vladimir Kolesnikov
Georgia Tech
kolesnikov@gatech.edu

ABSTRACT

Zero-Knowledge (ZK) proofs (ZKP) are foundational in cryptography. Most recent ZK research focuses on non-interactive proofs (NIZK) of small statements, useful in blockchain scenarios. Another line, and our focus, instead targets proofs of large statements that are useful, e.g., in proving properties of programs in ZK.

We specify a **zero-knowledge processor** that executes arbitrary programs written in a simple instruction set, and proves in ZK the correctness of the execution. Such an approach is well-suited for constructing ZK proofs of large statements as it efficiently supports complex programming constructs, such as loops and RAM access.

Critically, we propose several novel ZK improvements that make our approach concretely efficient: (1) an efficient arithmetic representation with conversions to/from Boolean, (2) an efficient read-only memory that uses $2 \log n$ OTs per access, and (3) an efficient read-write memory, BubbleRAM, which uses $\frac{1}{2} \log^2 n$ OTs per access. BubbleRAM beats linear scan for RAM of size > 3 elements! Prior ZK systems used generic ORAM costing orders of magnitude more.

We cast our system as a garbling scheme that can be plugged into the ZK protocol of [Jawurek et al, CCS'13].

Put together, our system is concretely efficient: for a processor instantiated with 512KB of main memory, each processor cycle costs 24KB of communication. We implemented our approach in C++. On a 1Gbps LAN our implementation realizes a 2.1KHz processor.

CCS CONCEPTS

• **Security and privacy → Cryptography**.

KEYWORDS

Zero Knowledge, Garbling Scheme, Verifiable Garbled Circuits

ACM Reference Format:
David Heath and Vladimir Kolesnikov. 2020. A 2.1 KHz Zero-Knowledge Processor with BubbleRAM. In *2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3372297.3417283

1 INTRODUCTION

Zero-Knowledge (ZK) proofs (ZKP) allow an untrusted prover P to convince an untrusted verifier V that a statement is true while revealing no other information. ZKP are fundamental cryptographic tools that are useful in a variety of settings. Recently, blockchain applications have encouraged intense research focus on succinct proofs of small statements. This focus has left proofs of large statements less explored, especially in concrete terms. Large statements are useful; for example, they capture properties of programs such as 'this program has a bug'.

1.1 Contribution

We construct a concretely efficient Zero-Knowledge processor, well-suited for proving large statements. It incorporates several technical contributions:

(1) BubbleRAM, a novel ZK oblivious RAM (ORAM), of amortized cost $\frac{1}{2} \log^2 n$ Oblivious Transfers (OT) per access. Critically, OTs for all accesses can run in parallel, resulting in constant round complexity. BubbleRAM is best for tiny arrays too, beating linear scan for RAM sizes > 3 elements!
(2) ZK oblivious ROM of amortized cost $2 \log n$ OTs per access.
(3) Arithmetic representation for authenticated values with efficient support for both arithmetic operations and converting to/from Boolean. Its efficiency is essential in memory operations, allowing for quick oblivious reshuffles.

We implemented our system in C++. On a 1Gbps LAN, our implementation realizes a 2.1KHz processor.

Our system proves bugs in generalized code snippets from [24] (e.g., we allow unbounded loops, which was fundamentally unsupported in [24]) and achieves comparable performance. For programs that exercise arbitrary memory access, we improve on [24] by sevearls orders of magnitude. Prior works implementing von Neumann architectures focused on NIZK [9, 11]; our total proof time is better by ≈ 3 orders of magnitude or more than their reported numbers.

Outline. We present our processor, arithmetic representation, ROM and RAM both informally (Sections 1.4 and 4) and in technical detail (Section 6). Security proofs are in Section 7. We discuss our implementation in Section 8 and evaluate its performance in Section 9.

1.2 High Level Intuition

The prover P can precompute in cleartext all intermediate values appearing during the computation of the processor. We allow P to choose message authentication codes (MACs) for all intermediate values via (many) OTs: each of her MACs is a proof that she has a particular value on a particular processor "wire".

P's proof task is to demonstrate that all of the MACs that she chose are related to one another by the processor's semantics. She does so by performing simple algebra, enabled by our MAC representation's support for communication-free homomorphic operations. This algebra allows P to construct a large collection of MACs that each authenticate the value 0. If P cheats in her OT selection and takes a MAC that does not correspond to the cleartext

CCS '20, November 9–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7089-9/20/11…$15.00
https://doi.org/10.1145/3372297.3417283

processor execution, she will be unable to construct a MAC of 0. P sends as proof a digest (computed by a hash) of all of these 'proofs of zero' to the verifier V, who precomputes the his own copy of the hash and checks the two are equal. Because the sent MACs each encode 0 regardless of P's input, the transmitted digest conveys no information to V: the protocol is Zero Knowledge.

1.3 ZKP from Garbling Schemes via JKO [27]

We cast our system as a garbling scheme (GS) [4]; it becomes a malicious-verifier ZKP system when used with the JKO framework [27]. It keeps the constant round complexity of JKO. Thus, we construct an efficient GS that satisfies the preconditions required to interface with [27]: correctness, soundness and verifiability.

1.4 Zero Knowledge Processor Architecture

Our central design choice is to implement a processor architecture. This choice is not standard: most ZK tools focus on direct circuit representations (either Boolean or arithmetic). In a direct circuit representation, program operations are implemented as circuits (for example, addition might be built from full 1-bit adders that are respectively built from AND and XOR gates), and then an overall circuit is wired together from these smaller circuits. Direct circuit representations are often efficient: small program circuits can be tuned to minimize computation and/or communication.

However, circuit representations have one significant downside: circuits do not adequately represent control flow. A circuit must implement all control paths in the program; thus, a naïve direct translation may result in exponential circuit size. Recent research has shown how to avoid the *communication* overhead of conditional branching [24], but the core *representational* problem remains. In particular, [24] does not scale to large numbers of execution paths. In contrast, typical programming models assume that the underlying system need only execute the taken control path.

An alternative to direct circuit representations, which we adopt, is to implement programs from arbitrarily composable building blocks. Each building block, implemented as a circuit, evaluates a single cycle of a low-level processor. More precisely, building blocks conditionally implement all instructions in a target instruction-set-architecture (ISA)[1]. The circuit (1) fetches the next instruction from memory, (2) decodes the instruction to determine which operation to perform, and (3) executes the required operation to update processor state. To execute an end-to-end program, we glue together many such cycles. The benefit of an indirect ISA representation is that the number of required building blocks is proportional to the cleartext running time, regardless of control flow.

This approach leads to a significant problem that must be addressed: the building block models all instructions and accesses memory, and hence incurs corresponding communication and computation costs. This work's technical goal is to significantly reduce these costs. We focus on the following subproblems:

- Each cycle reads an instruction from memory. Thus, we implement a read-only memory (ROM) suitable for holding instructions. The ROM uses only $2 \log n$ OTs per read.
- Each cycle implements the possibility of reading/writing to main memory. RAM access in the oblivious setting motivates research on Oblivious RAM (ORAM). We design and implement a custom ZK ORAM, `BubbleRAM`, which incurs no extra rounds of communication and uses only $\frac{1}{2} \log^2 n$ OTs per access.

The RAM and ROM introduce a residual problem: both are based heavily on the *oblivious permutation* of values, but unfortunately permuting a value encoded in a Boolean representation incurs overhead proportional to the number of bits in the value. Thus, we introduce and implement an *arithmetic* representation that is friendly to permutation. Our RAM/ROM store cells with 32-bit values and thus the arithmetic representation gives a factor 32 improvement in RAM/ROM operations. The arithmetic representation also allows efficient arithmetic (e.g. addition and multiplication), while supporting efficient Boolean operations. While our focus is constructing an efficient ZK processor, our subcomponents, especially our arithmetic representation and `BubbleRAM`, may be of independent interest.

2 RELATED WORK

We build an efficient ZK technique for proving large statements encoded in a low-level instruction set. In our review of related work, we focus on concretely efficient protocols.

ZK. ZK proofs [21, 22] are fundamental cryptographic primitives. ZK proofs of knowledge (ZKPoKs) [3, 16, 22] allow a prover to convince a verifier, who holds a circuit C, that the prover knows an input, or *witness*, w for which $C(w) = 1$.

Early practical ZK protocols, motivated by signatures and identification schemes, focused on algebraic relations, e.g. [15, 36]. More recently, ZK research has shifted focus to proofs of arbitrary statements: our work is in this more recent line. We next highlight works that prove arbitrary statements.

ZKP from garbling schemes. The most closely related works allow an interacting prover P to convince V of a satisfying assignment to a circuit by evaluating a Garbled Circuit (GC) [18, 24, 27]. [27] was the first work to construct concretely efficient proofs of arbitrary circuits. It also establishes a garbling scheme (GS)-based ZKP *framework* relying on *verifiable* GS: by satisfying a few GS requirements, new schemes can be plugged into [27]'s protocol to obtain malicious-verifier ZK. [18] improved the [27] framework and proved that common GC techniques are compatible.

Although our work is not immediately recognizable as a GC or GS technique (we use a custom algebraic representation, not standard GC), it fits neatly in the [27] framework. Hence, we review [27] in Section 3 as background to our approach.

[24] modernized verifiable GS by showing an efficient technique for circuits that include conditional branching, e.g. as the result of `if` program statements. Their technique's communication cost scales with the longest execution path, not with the size of the circuit. In addition, [24] presented a motivating use-case for verifiable GCs: proving the existence of a program bug in ZK. The authors argue that the community's intense focus on NIZK, motivated largely by blockchain applications, has left (interactive) proofs of larger

[1] This case of conditional branching does not lead to control-flow blow-up: the number of instructions is a small constant.

statements less explored. Verifiable GSs scale elegantly to arbitrarily sized proofs, and so are well suited to the larger statements resulting from the ZK bugs use-case.

Our work is similarly well-suited for this use case: our costs scale linearly with the program execution time. In Section 9 we compare to [24] wrt their ZK bugs use case. Our evaluation demonstrates that while [24] is slightly faster than ours for small, simple programs, (1) our approach is more general, since it need not explicitly annotate loops with hard-coded upper bounds and (2) our approach scales to more realistic programs due to our efficient RAM representation.

Succinct and non-interactive ZK. Ishai et al. [26], introduced the 'MPC-in-the-head' paradigm: here, P emulates in her head an MPC evaluation of the proof statement among virtual players. V inspects random portions of the evaluation transcript and thus gains confidence that the prover has a witness. By allowing V to inspect transcripts of only some virtual players, the protocol protects P's secret. This groundbreaking work spurred a flurry of subsequent MPC-in-the-head advances [1, 13, 20, 28].

Succinct non-interactive arguments of knowledge (SNARK) techniques construct extremely small proofs with fast verification time [7, 14, 19, 23, 34]. Early SNARKs require the use of a semi-trusted party, and thus more recent works have developed STARKs (succinct transparent arguments of knowledge) [5]. STARKs do not require trusted setup and rely on more efficient primitives.

[24] extensively compares verifiable garbling schemes with many of the works above: namely [28], Ligero, Aurora, Bulletproofs, STARK, and Libra [1, 6, 8, 12, 28, 42]. Their analysis demonstrates that while these works have excellent ZK performance in certain settings (e.g., small proof size, fast verification time, non-interactivity), they struggle to handle large proofs motivated by problems like the ZK bugs use-case. Thus, we focus our comparison on [24], whose focus (and interactivity) is similar to ours.

ZK Processors. A number of works also implement ZK processors [7, 10, 11]. These works build succinct, non-interactive proof engines. Thus, in a sense these approaches are more general than ours: our approach is interactive and requires proportional work from both the prover and verifier. The trade-off is efficiency. These works yield processors that run in the 10Hz range and hold only hundreds of bits of memory. In contrast, our processor operates in the KHz range and manipulates hundreds of kilobytes of memory.

ORAM. A key contributor to the efficiency of our processor is a novel RAM built for ZK. To the best of our knowledge, no prior work has built RAM with this explicit use case in mind. Instead, prior ZK works that interface with large RAM use standard oblivious RAM (ORAM) as a black box, e.g. [25, 33]. Thus, we provide performance comparison to state-of-the-art concrete-efficiency ORAMs. We examine 'Floram', a recent ORAM that scales well to large memory sizes [17], and a recent state-of-the-art square-root ORAM that is preferable for smaller memory sizes [35]. Comparing to these approaches is somewhat of an apples-to-oranges comparison: these approaches are generic ORAM tools well-suited to a number of scenarios. We compare to demonstrate the relative efficiency of `BubbleRAM`. Our analysis (Section 9) shows that `BubbleRAM` is orders of magnitude faster than the state-of-the-art ORAMs. Furthermore, `BubbleRAM` is directly embedded in a circuit and hence does not require extra rounds of communication to perform RAM lookup.

MPC Processors. A number of works build processors in the MPC setting. [32] were the first to formalize how a GC can interface with an ORAM in their Garbled RAM construction. While groundbreaking, Garbled RAM is a theoretical construction and to the best of our knowledge no implementations exist. More recently, [37, 41] execute machine code, implementing a MIPS processor as a Boolean circuit. This approach achieves impressive performance, but suffers from the high cost of memory and arithmetic operations. Our work also achieves a processor, but does so in the ZK setting. We take advantage of this more relaxed setting to achieve high performance, in particular with respect to RAM access.

3 ZKP FROM GARBLING SCHEMES

Jawurek et al. [27] based the first efficient arbitrary-ZK construction on GS. Prior to this breakthrough, ZK research focused on proofs of algebraic statements. Our system is built on OT, not on GC, but we can naturally cast it as a GS and reuse the [27] machinery. We review [27], for simplicity casting it as GC-based, and keeping in mind that [27] works with more general schemes:

In a GC protocol, one player, the generator, *encrypts* the circuit and sends the encryption to the second player. This second player, the evaluator, evaluates this encrypted circuit gate by gate under encryption. Finally, the players jointly decrypt the output. There are two properties of GC that are critical for ZK:

(1) GC protocols produce **authentic** output. That is, even a malicious evaluator cannot construct an output that successfully decrypts except by running the protocol as specified. Authenticity is a formal property of all garbling schemes [4].
(2) GC protocols hide intermediate values from the circuit generator. Indeed, after the generator sends the circuit encryption, he receives no messages from the evaluator until it is time to decrypt the output.

[27] builds a ZK construction on top of GC as follows:

V and P agree on a proof statement encoded as a Boolean circuit, C. Together, they run a GC protocol, where V acts as the generator and P acts as the evaluator. Since C encodes a ZK statement, only P provides input and C has only one output wire.

V encrypts C and sends the encryption to P. A malicious V can cheat by sending an invalid circuit encryption that leaks P's inputs. Therefore, [27] requires the following steps:

(1) P evaluates the GC and commits to the output encryption.
(2) V sends to P all randomness used to encrypt the circuit.
(3) P re-encrypts C to check that V's encryption is valid.
(4) P opens her commitment to the output.

Because of the added check, even a malicious V cannot cause P to open a commitment to an output computed on an invalid circuit garbling. Further, GC properties protect P's private input and ensure that the protocol is ZK: V learns nothing except that C outputs 1.

[27] established and [18] subsequently updated a *framework* for verifiable GSs. A verifiable GS must (1) satisfy Bellare et al.'s definitions of **correctness** and **authenticity** (renamed **soundness** in the ZK setting) [4] and (2) be **verifiable**, which informally allows P to check that V did not cheat. [27] provides a ZK protocol generic

to all verifiable GSs. While our approach does not manipulate GC, it fits cleanly in the verifiable GS framework. Thus, we formalize our work in this framework and rely on [27]'s protocol. We provide their protocol as reference in Appendix E.

4 TECHNICAL OVERVIEW

We construct a *highly generic* and *efficient* ZK proof engine. Since generality is a core goal, we designed our solution as a ZK *processor*. To support it, we construct an efficient ZK ROM and RAM, as well as an arithmetic representation that efficiently supports both arithmetic and Boolean operations. In this overview section, we informally present the core ideas of our constructions.

Arithmetic Representation for Verifiable Garbling. We implement the individual processor cycle building blocks using an arithmetic circuit representation. In Section 6.5 we present the representation in technical detail; we discuss it at a high level here. Recall that in the interactive ZK setting, the two players are the Verifier V and the Prover P. Recall also that only P has input and hence P can precompute all intermediate values in the program.

At a high level, our scheme ensures that for each intermediate circuit wire (1) P holds the cleartext value on the wire and (2) P and V hold additive shares of a message authentication code (MAC) on the wire. These MACs have two high level attributes:

- MACs are 'unforgeable' in the sense that P cannot efficiently find a MAC corresponding to a different cleartext value on the same wire. Thus each MAC proves that P has a particular value on a particular wire. Critically, the MAC on the program output wire thus authenticates the output value.
- To operate on arithmetic shares (e.g. to multiply), P asks V for additional input via OT. MACs are used to ensure that P asks for the correct input (i.e. does not cheat).

In more detail, let q be a large prime; the following arithmetic is done modulo q. V holds a secret uniform value $\Delta \in \mathbb{Z}_q$ that is global to the protocol. For each wire plaintext value $a \in \mathbb{Z}_q$, P holds a MAC share $a \cdot \Delta - A$ and V holds A, where $A \in \mathbb{Z}_q$ is also a secret drawn by V. A acts as a mask on P's share: since P knows neither A nor Δ, she cannot with high probability construct a MAC corresponding to a different value, say $(a+1) \cdot \Delta - A$.

Implementing Operations. The arithmetic representation features additive homomorphism, and thus addition, subtraction, and multiplication by a public constant require no communication. Other operations are implemented by OT:

As an example, suppose the players wish to decompose an arithmetic value into binary, where each bit is itself authenticated by an arithmetic MAC. That is, they wish to convert a 32-bit value into 32 1-bit values. To do so, P simply asks for the binary decomposition via OT. V's inputs to each OT are the two possible MACs for the corresponding bit. Thus P receives a MAC on each bit. Of course, P could cheat and ask for an arbitrary collection of bits. P proves she did not cheat as follows: First note that the binary MACs can be combined together using addition and multiplication by a public constant to form a new MAC of the decomposed value. P subtracts this combined MAC from her original MAC. If P did not cheat, then the result is a MAC of the value 0. P presents this MAC to V as proof that she did not cheat in the OTs. This MAC maintains ZK, because V receives the same message regardless of the decomposed value. In our approach, P constructs many of these 'proofs of 0' (`zerosActual`, MACs that authenticate a 0 value). For efficiency, P does not send these proofs individually, but instead accumulates them and at the end of the protocol sends a single hash of all proofs. V checks this hash against her own `zerosExpected` and is convinced P has a witness if they match.

Arithmetic MACs support other operations, such as multiplication. Perhaps the most significant operation is multiplication where one multiplicand is known to be either 0 or 1, which is achieved with only a single OT. This low cost is important because this binary multiplication is used extensively in permuting the processor's ROM and RAM. We discuss arithmetic in detail in Section 6.5.

Round Complexity . The arithmetic representation relies heavily on receiving inputs via OT. However, it *does not* incur multiple rounds of communication. This is because we ensure that all of P's OT inputs are her cleartext values. Thus P precomputes all inputs at the beginning of the protocol and issues all OT inputs at once. Our round complexity is the same as the underlying protocol of [27].

ROM. The ZK processor stores program instructions in a read-only memory (ROM) such that on each cycle the processor can obliviously read the next instruction.

Our ROM is based on *oblivious permutation* [38]. At a high level, we allow P to freely permute the ROM, and then check that that the indexes she accesses are consistent with the provided index value. Because P can predict, by executing in cleartext, the state of the processor over time, she can arrange the ROM favorably such that the elements are in the order that they are needed. Additional work is needed to ensure that instructions can appear more than once, but this modification is easily achieved by a clever circuit construction in [31]. Altogether, each ROM access incurs only $O(\log n)$ amortized communication. The construction is described in Appendix C.

`BubbleRAM`. The ZK processor stores the registry and main memory in practically efficient ZK ORAMs. Like our ROM, `BubbleRAM` is based heavily on oblivious permutation. P freely permutes the RAM over time, such that elements that will be accessed soon are close to the front of the RAM (elements *bubble up*). That is, a given element's distance from index 0 is related to the number of RAM accesses before that element is next needed. By continually rearranging the RAM, P ensures that each access need only look at index 0. The RAM incurs amortized $\frac{1}{2}\log^2 n$ communication per access. The construction is described in Section 6.8.

5 NOTATION

- $x \triangleq y$ denotes that x is defined as y.
- $\mathbb{Z}_n$ denotes the integers modulo n.
- σ is the statistical security parameter (e.g. 40).
- H is a collision resistant hash function.
- V is the verifier. We refer to V as he, him, his, etc.
- P is the prover. We refer to P as she, her, hers, etc.
- $\langle x, y \rangle$ denotes a pair where V knows x and P knows y.
- $a \equiv_n b$ denotes that a is congruent to b modulo n.
- We denote the type of arrays with elements of type $type$ and fixed length n by $[type\ ;\ n]$. We initialize a fixed size array where each index is filled with a value x by writing $[x\ ;\ n]$.

6 TECHNICAL APPROACH

Our contribution is a generic and practically efficient Zero Knowledge processor that proves statements encoded as programs in a small instruction set. In this section, we present our construction, ZKM (ZK Machine), in technical detail.

We formalize our approach as a *verifiable garbling scheme* [4, 18, 27]. Garbling schemes are typically understood as tools used to construct GC protocols, and so our approach at first glance may seem incompatible. However, on closer inspection our approach does fit with the formal notion of garbling.

A verifiable garbling scheme is a 6-tuple of algorithms:

$$(\mathsf{ev}, \mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De}, \mathsf{Ve})$$

At a high level, ev provides the cleartext semantics for the construction: i.e., it specifies whether or not P's private input causes the program to output 1. Gb, En, Ev, and De allow the two players to achieve the same result in an honest verifier ZK protocol. In particular, Gb specifies how V sets up all of his OT inputs[2], En specifies which OT outputs P receives based on her input, Ev specifies how P constructs her authentic output, and De specifies how V checks that P's output is correct. Finally, Ve facilitates *malicious* verifier ZK protocols: it specifies how P checks that all messages received from V were constructed correctly.

Construction 1. ZKM *is a privacy-free*[3] *verifiable garbling scheme expressed as the following 6-tuple:*

$$(\mathsf{ZKM.ev}, \mathsf{ZKM.Gb}, \mathsf{ZKM.En}, \mathsf{ZKM.Ev}, \mathsf{ZKM.De}, \mathsf{ZKM.Ve})$$

All algorithms except En and De are based on running a fixed number T of processor cycles (runtime T is agreed on in advance):

(1) ZKM.ev executes processor steps in cleartext. After T steps, ev outputs whether the processor's first register holds the value 1.
(2) ZKM.Gb garbles individual processor steps and, as it goes, generates the MACs that are V's OT inputs. The algorithm also accumulates the expected code, zerosExpected, that V expects to receive from P.
(3) ZKM.En specifies which MAC input values P should receive as a function of her input. In our scheme, all of P's inputs are binary choices implemented by 1-out-of-2 OT. Formally, ZKM is a projective garbling scheme. Thus ZKM.En is a simple mapping: It specifies that if P's ith input bit is 0 she receives the 0 OT secret, else the 1 OT secret. For reference, an algorithm for ZKM.En is given in Appendix F.
(4) ZKM.Ev executes processor steps using P's MAC values and accumulates authentication codes that are to be sent to V.
(5) ZKM.De is a straightforward comparison of the MACs expected by V and sent by P. If they match, the algorithm outputs 1, else 0. A specification of ZKM.De is given in Appendix F.
(6) ZKM.Ve allows P to rerun ZKM.Gb to reconstruct the messages received from V. Thus, P can check that all V's messages were correctly constructed.

In the remainder of this section, we describe ZKM.ev, ZKM.Gb, and ZKM.Ev in detail. Section 7 formalizes the security of ZKM (proofs are in Appendix A). In particular, ZKM satisfies the security definitions specified in [18]. Theorems in Section 7 imply the following:

Theorem 6.1. ZKM *is* ***correct***, ***sound***, *and* ***verifiable***.

This fact, combined with theorems from [27], implies:

Theorem 6.2. *The protocol π_{ZK} of [27] instantiated with* ZKM *is a secure protocol that achieves malicious verifier Zero Knowledge.*

6.1 Cleartext Processor Specification

We start by explaining the cleartext specification, ZKM.ev. In many garbling schemes, ev is implicit: typical schemes operate over circuits with a small fixed set of gate types and their semantics are clear. In contrast, our approach features a relatively complex processor architecture. Therefore, we carefully specify ZKM.ev such that we have a reference against which to check correctness.

Our processor is similar to a hardware processor and is a state machine that includes the following components:

- A small collection of registers that hold local memory.
- A large main memory.
- The program text, held in a read-only memory (ROM).
- A program counter that indexes the ROM.
- A collection of the prover's private input.

The core task in specifying cleartext semantics is to show how the processor steps from one state to the next.

We thus formalize program state. Our machine manipulates 32 bit integers, a typical choice in cleartext machines. We define the space of *values* (i.e., the types of objects held by memory and registers and manipulated by low-level operations) as 32 bit integers:

$$v \triangleq \mathbb{Z}_{2^{32}}$$

The low-level operations of the machine primarily focus on manipulating *registers*, each of which hold a single value. We choose to implement 32 registers and refer to the registers collectively as the *registry*. We formalize a registry as an array of 32 values:

$$\mathtt{registry} \triangleq [v\ ;\ 32]$$

The processor manipulates the registry and the main memory by executing program instructions. On each state transition, the processor reads a single instruction from the program and performs the indicated state transformation. An instruction is a 4-tuple of an *op-code* and 3 arguments. Formally, let $src_0, src_1, tar \in v$. The space of instructions is defined:

$$\mathtt{instr} \triangleq (\mathsf{op}, src_0, src_1, tar)$$

The space of program op-codes is a finite list of symbols:

op ≜ add | mul | lt | ⋯	arithmetic/comparisons on registers
| beqz	branch to src_1 if src_0 is zero
| load	load memory at src_0 to register tar
| store	store register src_0 to address src_1
| input	read P's input value into register tar

We include op-codes that perform register arithmetic/comparisons, that conditionally branch, that read/write main memory, and that receive input from P. As an example, on an add instruction the

[2] In typical garbling schemes, Gb provides *encryptions* of programs. Our scheme features no such encryptions, so we accordingly simplify our presentation by adjusting the [4] notation to omit program encryptions.
[3] Privacy-free schemes do not protect V's input (V has no input in the ZK setting). Thus our approach is not suited for the secure 2PC setting.

processor (1) reads register src_0, (2) reads register src_1, (3) adds together these two values, and (4) writes the result to register tar.

A *program text* is an indexed collection of instructions. On each cycle, the processor reads from this collection based on the *program counter*. Formally, a program counter is a value and a program text is an array of instructions:

$$\texttt{pc} \triangleq v \qquad \texttt{text} \triangleq [\texttt{instr}]$$

While small amounts of memory can be stored in the registry, complex programs require large amounts of space. Thus, the processor includes a large *main memory*. Programs move values into and out of main memory using `load` and `store` instructions. Formally, a *memory* of size $space \in \mathbb{Z}_{2^{32}}$ is an array of values of size $space$:

$$\texttt{memory}(space) \triangleq [v\ ;\ space]$$

Finally, we specify how the processor loads in P's private input. The `input` op-code instructs the processor to read in a single value from P's input. Since P might have more than 32 bits of input, we allow the processor to read in each of P's values one by one. Formally, the input is a read-only *stack* of values:

$$\texttt{inp} \triangleq \texttt{stack}(v)$$

We assume read-only stacks come equipped with a method pop that pops the top of the stack and returns the popped value.

We now formalize processor state. The state fully specifies all information needed to perform each instruction. Formally, a state with memory size $size$ is a tuple of (1) a program, (2) a program counter, (3) a registry, (4) a memory, and (5) an input stack:

$$\texttt{state}(size) \triangleq (\texttt{text}, \texttt{pc}, \texttt{registry}, \texttt{memory}(size), \texttt{inp})$$

We define the semantics of our language in terms of a *stepping* operation. That is, `step` (Figure 1) is a procedure that mutates the program state according to the current instruction.

A *program* is a 3-tuple of (1) a program text, (2) a *space* parameter (i.e., the size of the main memory), and (3) a *time* parameter. Formally, let $space, time \in \mathbb{Z}_{2^{32}}$:

$$\texttt{program} \triangleq (\texttt{text}, space, time)$$

We repeatedly step to run a program $prog$ on P's private input I:

```
eval(prog, I) :
  (T, space, time) ← prog
  ▷ initialize the state with the program text,
  ▷ P's input, and an empty registry/memory.
  s ← (P, 0, [0 ; 32], [0 ; space], I)
  for i ∈ [0..time) :
    step(s)
  return s.registry[0] == 1
```

We establish a convention that program execution constitutes a valid proof if register 0 contains 1 at termination.

6.2 The Authenticated Processor

Now that we have established cleartext semantics, we specify the *authenticated* processor which performs the same task. Our goal is to specify `ZKM.Gb` and `ZKM.Ev`, the actions respectively taken by V and P. The details of these two procedures are *nearly identical*. Only

```
step(s) :
  ▷ Decompose the state into its parts.
  (T, pc, R, M, I) ← s
  ▷ Read the instruction from the program text.
  (op, src0, src1, tar) ← T[pc]
  ▷ Read from the source registers.
  arg0 ← R[src0]
  arg1 ← R[src1]
  ▷ Conditionally dispatch on the op-code.
  switch(op) :
    case(add) : R[tar] ← arg0 + arg1
    case(mul) : R[tar] ← arg0 · arg1
    case(lt) : R[tar] ← arg0 < arg1
    ...   ▷ Other arithmetic/comparisons.
    case(load) : R[tar] ← M[arg0]
    case(store) : M[arg1] ← arg0
    case(input) : R[tar] ← I.pop()
  ▷ Update the program counter.
  if op == beqz ∧ arg0 == 0
    then pc ← arg1
    else pc ← pc + 1
```

Figure 1: `step`, the core of the cleartext specification for the authenticated processor. `step` takes as an argument and mutates a program state. To perform a step of computation, the processor (1) reads an instruction, (2) reads the registry, (3) performs computation according to the op-code, including possibly reading/writing main memory, and (4) (possibly) writes the resulting value to the registry.

the low-level operations (e.g. how to multiply two values) differ. Thus, we begin our discussion at a higher level, defining authentic constructions that both P and V use. Later, we differentiate the actions of the two players.

The following definitions closely reflect the specification in Section 6.1. We distinguish authentic types from their cleartext variants by marking them with a hat. The differences between the authentic specification and its cleartext variant are two-fold: in the authentic setting (1) it is infeasible for P to forge intermediate values and (2) V's view is independent of P's input.

The authenticated processor manipulates *authentic* values. We discuss the details of authentic values carefully in Section 6.5. For now, assume that there exists a type $\hat{v}$ that encodes an integer in a field $\mathbb{Z}_q$ where q is a σ-bit prime. We assume (and later show) that this type supports algebraic operations, comparisons, and an operation `mod32` which computes mod 2^{32}.

Like cleartext instructions, authentic instructions are 4-tuples of an op-code and arguments. Authentic op-codes are simply authentic values. Formally, let $src_0, src_1, tar \in \hat{v}$:

$$\widehat{\texttt{op}} \triangleq \hat{v} \qquad \widehat{\texttt{instr}} \triangleq (\widehat{\texttt{op}}, src_0, src_1, tar)$$

To read instructions in the authenticated setting, we use an authentic read-only memory. We defer the discussion of the authentic ROM to Appendix C. For now, assume that there exists a type $\widehat{\texttt{rom}}$

and that for a ROM of size n there exists (1) a procedure `initROM` which initializes the ROM from a cleartext array and (2) a procedure `readROM` that returns the element stored at an index specified by an authentic value. An *authentic program text* is an encrypted read-only memory of instructions. We use an authentic program counter to index the program text:

$$\widehat{\texttt{text}} \triangleq \widehat{\texttt{rom}}(\widehat{\texttt{instr}}) \qquad \widehat{\texttt{pc}} \triangleq \hat{\nu}$$

To represent the registry and main memory, we use an authentic RAM that we specify in Section 6.8. For now, assume that there exists a type $\widehat{\texttt{ram}}$ equipped with the following two procedures: (1) `initRAM` which initializes a RAM of a specified size filled with 0s, and (2) `accessRAM` which reads from and optionally (based on an authentic flag argument) writes to a RAM index. An *authentic registry* is an authentic RAM of 32 values, and an *authentic memory* is an authentic RAM of a parameterized number of values:

$$\widehat{\texttt{registry}} \triangleq \widehat{\texttt{ram}}(\hat{\nu}\ ;\ 32) \qquad \widehat{\texttt{memory}}(\mathit{space}) \triangleq \widehat{\texttt{ram}}(\hat{\nu}\ ;\ \mathit{space})$$

Finally, we represent P's input. It turns out that the handling of P's input is perhaps the most complicated digression from the cleartext specification. For now, assume that there exists an efficient algorithm `readInput` that takes an authentic flag as an argument. If the flag encodes 1, then P's next 32 bit input is returned as an authentic value, and otherwise the authentic value 0 is returned. We discuss the subtleties of P's input in Section 6.4.

With the subcomponents defined, we can specify authentic state. An *authentic state* of size *size* is a tuple of an authentic program text, program counter, registry, and memory.

$$\widehat{\texttt{state}}(\mathit{size}) \triangleq (\widehat{\texttt{text}}, \widehat{\texttt{pc}}, \widehat{\texttt{registry}}, \widehat{\texttt{memory}}(\mathit{size}))$$

With the definitions we have, we specify the authentic variant of `step`, $\widehat{\texttt{step}}$ (Figure 2). Again, $\widehat{\texttt{step}}$ summarizes the actions taken by both V and P in order to run a proof. The key difference between `step` and $\widehat{\texttt{step}}$ is (1) that $\widehat{\texttt{step}}$ uses the authentic variants of values, value operations, and data structures and (2) that $\widehat{\texttt{step}}$ does not conditionally dispatch, but instead performs all operations and uses algebra to select the result. Informally, because $\widehat{\texttt{step}}$ performs all operations, V learns nothing about P's input by running the steps.

6.3 Processor Modes

Thus far, we have presented $\widehat{\texttt{step}}$ as a generic description of both V's and P's actions. However, the low-level operations that $\widehat{\texttt{step}}$ uses are *different* depending on the player. For example, V multiplies two authentic values differently than P. In this section, we set up an infrastructure that formalizes the differences between these actions.

Our key tool is a *mode* variable, which is inspected to decide which low-level actions should be taken. We specify that there is a global variable `mode` that can have one of following values:

$$\texttt{mode} \in \{\texttt{VERIFIER}, \texttt{PROVER}, \texttt{INPUT}\}$$

The `VERIFIER` and `PROVER` modes are respectively used by V and P to perform their respective tasks. The `INPUT` mode is used by P to convert her cleartext input into a list of OT selection bits. We discuss `INPUT` mode more in Section 6.4. In addition to `mode`, we specify a number of other global variables that the players need:

$\widehat{\texttt{step}}(s)$:

▷ Decompose the state into its parts.

$(T, pc, R, M) \leftarrow s$

▷ Read the next instruction from ROM.

$(op, src_0, src_1, tar) \leftarrow \texttt{readROM}(T, pc)$

▷ Read the inputs from the registry.

▷ 0 flags indicate these accesses are not writes.

$arg_0 \leftarrow \texttt{accessRAM}(R, 0, src_0, 0)$

$arg_1 \leftarrow \texttt{accessRAM}(R, 0, src_1, 0)$

▷ Read cell inp_0, and if opcode is store, write arg_1 to cell arg_0.

$m \leftarrow \texttt{accessRAM}(M, op == \texttt{store}, arg_0, arg_1)$

▷ Conditionally get next input value from prover

$inp \leftarrow \texttt{readInput}(op == \texttt{input})$

▷ conditionally assign val based on op

$val \leftarrow inp$

$val \leftarrow val + ((op == \texttt{add}) \cdot (arg_0 + arg_1))$

$val \leftarrow val + ((op == \texttt{mul}) \cdot (arg_0 \cdot arg_1))$

$val \leftarrow val + ((op == \texttt{lt}) \cdot (arg_0 < arg_1))$

$val \leftarrow val + ((op == \texttt{load}) \cdot m)$

$\cdots$ ▷ other operations

▷ Ensure val is in range of ISA values.

$val \leftarrow \texttt{mod32}(val)$

▷ Write the output val to the output register tar.

$R \leftarrow \texttt{accessRAM}(R, ((op \neq \texttt{beqz}) \cdot (op \neq \texttt{store})), tar, val)$

▷ if $op = \texttt{beqz}$ and $arg_0 = 0$, branch to arg_1; else proceed.

$pc \leftarrow pc + 1 + (((op == \texttt{beqz}) \cdot (arg_0 == 0)) \cdot arg_1 - (pc + 1))$

Figure 2: $\widehat{\texttt{step}}$, the authentic variant of `step` (Figure 1). $\widehat{\texttt{step}}$ is the high level specification of both V's and P's actions when running the authenticated processor. We emphasize that all values manipulated in this algorithm are *authentic* values.

- `otInputs` is a list of V's OT inputs: i.e., they are the values that P chooses between during OT. Formally, `otInputs` is a list of pairs of pairs of integers in $\mathbb{Z}_q$. For each entry in the list, P chooses between the left pair and the right pair.
- `otChoices` is a list of P's OT selection bits. This list is derived from the program and P's cleartext input (see Section 6.4).
- `otOutputs` is a list of P's OT outputs: i.e., they are the values in `otInputs` that P chose according to `otChoices`. Formally, `otInputs` is a list of pairs of integers in $\mathbb{Z}_q$.
- The eventual proof that P sends to V is a list of authentic values that each encode 0. Both players maintain a list of these authentic 0 values: V maintains a list `zerosExpected` while P maintains a list `zerosActual`. V is convinced when P can produce a value $H(\texttt{zerosActual}) = H(\texttt{zerosExpected})$.

We now formalize `ZKM.Gb` and `ZKM.Ev`. Both algorithms set the mode, set the initial state, and call $\widehat{\texttt{step}}$ *time* times (Figure 3).

`ZKM.Ve` allows P to check that messages sent by V were correctly constructed. `ZKM.Ve` is defined similarly to `Gb` and `Ev`. Specifically, `ZKM.Ve` is given access to all of V's OT inputs. The algorithm allows

ZKM.Gb(1^σ, *prog*) :

▷ set the global mode so that verifier actions are taken.

mode ← VERIFIER

$(T, space, time)$ ← *prog*

▷ Initialize the processor state.

s ← (initROM(T), 0, initRAM(32), initRAM(space))

for $i \in [0..time)$

$\widehat{\text{step}}(s)$

▷ Read the output from register 0

out ← accessRAM($s.\widehat{\text{registry}}$, 0, 0, 0)

▷ Check that the program output is 1.

zerosExpected.push($1 - out$)

return (otInputs, H(zerosExpected))

Figure 3: V's actions when running a program. P's actions in ZKM.Ev are extremely similar, so we postpone that algorithm to Appendix F. Both players set relevant global variables, call $\widehat{\text{step}}$ $time$ times, and compute a digest of the expected output/actual output.

P to re-garble in the same manner as Gb except that, instead of drawing random values, P draws values from V's OT inputs and checks they are consistent with low-level operations. For example, P checks that OT choices are correctly separated by a global Δ value. In formal detail, ZKM.Ve requires an additional CHECK mode. For brevity and clarity we omit CHECK mode from our presentation. The algorithm for ZKM.Ve is provided in Appendix F.

6.4 Expanding P's Inputs

Upon close inspection, there are two mismatches between our cleartext specification eval (Section 6.1) and the verifiable garbling scheme infrastructure of [27]:

(1) [27] requires the garbling scheme to be *projective*. In a projective scheme, P's input must be *binary*. This requirement allows a ZK protocol built on top of 1-out-of-2 OT. In eval, P's inputs are 32-bit integers, not individual bits. Thus we must transform P's input into its binary representation.
(2) In our scheme, P uses OT not only for her input, but also to perform low-level operations like multiplication (as we show shortly). To fit with the [27] framework, we formalize these 'auxiliary' OTs as part of P's binary input. I.e, P's input not only includes her independently chosen values, but also many *dependent* bits used to compute low-level operations.

Thus, we provide a procedure that 'expands' P's input into all of her OT selection bits. It may on the surface appear that this is simply an extra step needed to interface with [27]. However, this same procedure also plays a critical role in implementing our scheme: for efficiency, we precompute all OT inputs such that all OTs can be completed in constant rounds.

This 'input expansion' motivates the inclusion of INPUT mode. To expand her input, P runs the authentic processor in INPUT mode, storing her selection bits in the list otChoices. Formally, this step happens as a preprocessing step by P and is outside the scope of our garbling scheme: from the perspective of our formal garbling scheme, P 'just knows' her expanded input ahead of time.

This subtlety is also significant to our formalism ZKM.ev, the cleartext specification. To be pedantic, ev must operate over this same expanded input. However, our cleartext specification defined in Section 6.1 instead uses only P's cleartext input encoded as a stack of 32 bit integers. Therefore, the *formal* cleartext specification ZKM.ev is identical to eval (Section 6.1) except:

- ZKM.ev discards unneeded auxiliary input bits.
- Upon an input instruction, ZKM.ev reads in 32 input bits and composes them into a 32 bit value.

Therefore, ZKM.ev performs the same computation on P's expanded input as eval performs on P's unexpanded inputs.

6.5 Arithmetic representation

We represent authentic values $\hat{v}$ as message authentication codes (MACs) of integers modulo q where q is a σ-bit prime. Let $a \in \mathbb{Z}_q$ be an arbitrary value. Let $A, \Delta \in \mathbb{Z}_q$ be uniform values drawn by V, except that Δ must not be 0. The MAC of a, $[\![a]\!]$ is as follows:

$$[\![a]\!] \triangleq \langle A, a \cdot \Delta - A \rangle$$

Recall, this notation indicates V holds A and P holds $a \cdot \Delta - A$. Therefore, the players hold additive shares of $a \cdot \Delta$. Δ is global to all MACs and is analogous to a Garbled Circuit free XOR offset of [30] and the follow-up arithmetic offset of [2].

Under this representation, addition is a homomorphism:

$$\begin{aligned}[\![a]\!] + [\![b]\!] &\equiv_q \langle A, a \cdot \Delta - A \rangle + \langle B, b \cdot \Delta - B \rangle \\ &\equiv_q \langle A + B, (a \cdot \Delta - A) + (b \cdot \Delta - B) \rangle \\ &\equiv_q \langle A + B, (a + b) \cdot \Delta - (A + B) \rangle \\ &\equiv_q [\![a + b]\!]\end{aligned}$$

Subtraction and multiplication by a public constant are similarly homomorphisms. Public constants can also be easily encoded: for a given constant c, the players use $[\![c]\!] \equiv_q \langle c \cdot \Delta, 0 \rangle$.

Other operations on authentic values are performed by having P and V perform OT. We explain these operations next.

6.6 Non-homomorphic operations

6.6.1 Boolean multiplication. We start by presenting multiplication where the left multiplicand is known to be a MAC of either 0 or 1 (Figure 4). We begin with this operation because it is simple and because its key ideas carry over to other non-homomorphic operations. Consider a multiplication of $[\![a]\!]$ by $[\![b]\!]$ where $a \in \{0, 1\}$. Informally, we use the fact that P *knows* the value a. Thus, P can locally multiply her share of $[\![b]\!]$ by a. Unfortunately, this does not line up with our representation. In particular, P now holds:

$$a \cdot (b \cdot \Delta - B) \equiv_q (a \cdot b \cdot \Delta) - (a \cdot B)$$

The term $a \cdot B$ is not a valid mask since V does not know (and must not learn) a. To account for this, we have V and P communicate via OT. Specifically, we allow P to choose a value $(a \cdot B) - C$ where C is a fresh mask: V allows P to choose between $0 - C$ and $B - C$ via OT. P chooses based on a and hence receives the desired value. The two players can now locally compute a valid MAC:

$$\langle C, (a \cdot b \cdot \Delta) - C \rangle$$

```
mul1([[a]], [[b]]) :
  ⟨a_V, a_P⟩ ← [[a]]
  ⟨b_V, b_P⟩ ← [[b]]
  ▷ a_P ≡_q a · Δ − a_V
  ▷ b_P ≡_q b · Δ − b_V
  switch(mode) :
    case(VERIFIER) :
      a'_V ←$ {0, q − 1}
      ab_V ←$ {0, q − 1}
      otInputs.push((0 − a'_V, 0 − ab_V), (Δ − a'_V, b_V − ab_V))
      zerosExpected.push(a_V − a'_V)
    case(INPUT) :
      otChoices.push(a)
    case(PROVER) :
      (a'_P, δ) ← otOutputs.pop()
      ▷ a'_P ≡_q a · Δ − a'_V
      ▷ δ ≡_q a · b_V − ab_V
      zerosActual.push(a'_P − a_P)
      ▷ a'_P − a_P ≡_q a_V − a'_V
      ab_P ← a · b_P + δ
      ▷ ab_P ≡_q a · (b · Δ − b_V) + a · b_V − ab_V
      ▷ ab_P ≡_q (a · b) · Δ − ab_V
  return ⟨ab_V, ab_P⟩
```

Figure 4: The special case MAC multiplication procedure where the first argument $a \in \{0, 1\}$. In the input phase, P uses a as an OT input. The OT has two functions: (1) it allows P to obtain δ which is used to compute the product and (2) it allows P to obtain a second copy of a MAC for a and thus prove she did not cheat in her OT input.

However, the protocol we have specified is not secure: there is no guarantee that P chose her OT input correctly! Therefore, we add extra values to V's OT inputs that allow P to prove her OT selection is made according to a. Specifically, V chooses another fresh mask A' and allows P to choose between the following pairs:

$$(0 - A', 0 - C) \qquad (\Delta - A', B - C)$$

Notice that when P selects based on a, she will receive $a \cdot \Delta - A'$: i.e., she will receive a new MAC for a. This allows her to authenticate her OT input to V: she computes $(a \cdot \Delta - A) - (a \cdot \Delta - A') \equiv_q A' - A$, i.e. a MAC of 0. She can present this MAC of 0 as proof that she selected her OT input according to the protocol.

Figure 4 formalizes the previous discussion as a procedure `mul1`. Note that P's tasks are broken into two phases: the `INPUT` phase where she provides her OT inputs and the `PROVER` phase where she computes her share and the zero MAC. `mul1` requires only 1 OT.

6.6.2 32-bit multiplication. `mul32` (see Appendix F) presents a 32-bit variant of multiplication. This implementation is more general than `mul1`, but requires more OTs. Our construction uses both variants to efficiently implement the processor specification. In $\widehat{\texttt{step}}$, we use overloaded notation: by $a \cdot b$ where $a, b \in \hat{v}$ we mean an operation which intelligently selects 1-bit multiplication if a can be statically deduced to hold either 0 or 1 (for example, a is the output of a comparison) and otherwise uses 32-bit multiplication.

The implementation of 32-bit multiplication is a natural generalization of 1-bit multiplication. Instead of choosing only 1 OT output, P must choose 32 OT outputs, each based on a bit in a. Like `mul1`, these OTs allow P to both construct a new MAC for a that authenticates her OT selection and to construct a MAC for $a \cdot b$.

6.6.3 Projection and comparisons. We implement comparisons on top of low-level Boolean operations (note that we can implement Boolean operations using 1-bit multiplication, addition, and subtraction). The implementation of comparisons is typical.

However, we must show how to convert an authentic value into its binary representation. `project` (see Appendix F for full algorithm) projects an n-bit value into n 1-bit values. The implementation of `project` is similar in flavor to both `mul1` and `mul32`. Specifically, let P's MAC share of a be $a \cdot \Delta - A$. P uses the bits of a as her n OT selection bits. As OT output, she receives a projection of a. She rebuilds this projection of a into a new MAC $a \cdot \Delta - A'$ using only homomorphic operations and then proves that this new MAC indeed encodes a by including $A' - A$ in her proof.

6.6.4 Mod 2^{32}. $\widehat{\texttt{step}}$ performs operations on authentic values that can cause them to escape $\mathbb{Z}_{2^{32}}$. Thus, after we perform all operations, we must clamp the output back to $\mathbb{Z}_{2^{32}}$ before writing it to the registry. `mod32` (see Appendix F) performs this operation. With `project` available, `mod32` is trivial: First, the procedure projects its argument a into 64 bits. Then, it reconstructs a MAC for the high 32 bits using homomorphic operations, subtracts this reconstruction from a, and outputs the clamped result.

6.6.5 Reading P's input. P's independent input (i.e. her proof witness) is read in by multiple OTs. Recall in our cleartext semantics, P's input is a stack of 32 bit numbers. `readInput` (see Appendix F) uses the top of this stack to choose 32 OTs.

`readInput` takes as an argument an authentic value that is either 0 or 1. This flag indicates if P 'actually' provides her independent input or not (recall, the processor attempts to read P's input every cycle, whether or not it is currently needed). If the flag is 0, then P does not pop her input stack and the procedure returns a MAC of 0.

6.7 $2 \log n$ ROM

For lack of space, we defer the full presentation of the ROM to Appendix C. The ROM is a relatively simple component based on an idea in [31] and that we briefly explain at a high level in Section 4.

6.8 `BubbleRAM`: $\frac{1}{2}\log^2 n$ RAM access

We now explain the construction that we use to represent both the registry and main memory: `BubbleRAM`. `BubbleRAM` features excellent concrete performance: each access costs amortized $\frac{1}{2}\log^2 n$ OTs. `BubbleRAM` is based on allowing P to look ahead at the access order and using this knowledge to permute memory.

A permutation can be achieved in a circuit using a Waksman permutation network [38]. Permutation networks are recursive constructions where the base case of permuting two elements is achieved using an individual 'swap' gate that conditionally swaps two elements. The full recursive construction includes $n \log n - \frac{n}{2}$

```
rearrange(t, array) :
  for i ∈ [log |array| − 1..0] :
    if (2^i | t) : partition(array[0..2^{i+1}], selection)
```

Figure 5: `rearrange` formalizes the strategy for rearranging a RAM whose elements are stored in $array$ before the access at timestep t. The procedure `partition` on an array of n elements and a set $selection$ of up to $\frac{n}{2}$ selected indexes permutes the array such that the selected elements appear in the first $\frac{n}{2}$ indexes. $selection$ is chosen by P, who precomputes the set based on her private input.

swap gates. Due to our algebraic representation, a swap gate of two 32 bit elements is implemented by a 1 bit multiplication based on P's private input and requires only a single OT. Specifically, to swap $x, y \in \hat{v}$ based on P's private bit b, the players compute:

$$\delta \leftarrow b \cdot (x - y)$$
$$\texttt{return } (x - \delta, y + \delta)$$

Thus, a permutation can be achieved by $n \log n - \frac{n}{2}$ OTs.

`BubbleRAM`'s key primitive is an oblivious *partition* on the first i RAM elements. An oblivious partition of size i allows P to select half of the first i elements in RAM and move them to the first $\frac{i}{2}$ slots in RAM. An oblivious partition is a special case of permutation, and hence can be implemented using $i \log i - \frac{i}{2}$ OTs [38].

A factor 2 more efficient algorithm permutes only the front half of the partition, then pairwise oblivious swaps elements in the front and back halves. This algorithm computes a partition on $2i$ elements (i.e. it moves i elements forward) using $i \log i + \frac{i}{2}$ OTs.

Informally, we describe elements that are needed 'soon' (i.e. within a small number of accesses) as 'hot' and elements that are not needed for many accesses as 'cold'. P repeatedly partitions memory such that hot elements tend to be close to the front of the array. Moreover, an element's distance from slot 0 is related to how hot it is: an element that is cold might be far from slot 0, an element that is warm will likely be close to slot 0, and the hottest element (i.e. the element needed next) *will* be in slot 0. The overall goal is to ensure that memory slot 0 holds the hottest element before each access. Thus, to perform each access, the players need only look at index 0. To maintain this 'temperature gradient', P uses her private bits to partition RAM.

Formally, `BubbleRAM`'s correctness invariant is as follows:

Invariant 1. *For all $i \in [0..\log n]$, at time-step t (i.e., after t memory accesses) the next $2^i - (t \mod 2^i)$ memory indexes to be accessed are located in the first 2^i RAM slots.*

Invariant 1 formalizes the 'temperature gradient'. An immediate corollary is that at each time-step, the hottest element is in slot 0.

To maintain Invariant 1, P partitions the memory before each access in order to move hot elements towards the front of RAM. Precisely, at each time-step t, for each $i \in [\log n - 1, \log n - 2, .., 0]$, P programs a partition of size 2^{i+1} if 2^i divides t. For example, at time-step 4, P programs a partition on the first 8 RAM elements, on the first 4 RAM elements, and on the first 2 RAM elements. At time-step 5, P programs a partition on the first 2 elements only.

Theorem 6.3 (RAM Correctness). *By applying `rearrange` (Figure 5) to the RAM at each timestep, P maintains Invariant 1. Hence, on each access slot 0 holds the next element to be accessed.*

Proof. By induction on time-steps. Informally, partitions are precisely the tool needed to maintain Invariant 1: a partition of size 2^{i+1} selects 2^i elements and moves them to first 2^i memory slots.

- Consider time-step 0, the induction base case. Because every integer divides 0, in this step P applies partitions of all sizes 2^{i+1} such that $2^i < n$. Thus, Invariant 1 is established.
- Suppose that Invariant 1 holds for time-step t. By partitioning the array according to `rearrange`, Invariant 1 also holds in time-step $t + 1$.
  Notice that $t + 1$ *overflows* some size 2^i. In particular, consider some i such that $2^i = t+1$. Invariant 1 in time-step t guarantees nothing about these first 2^i elements ($2^i - (t \mod 2^i) = 1$, and this 1 element might no longer be needed in step $t+1$). However, we do still know something about the first 2^{i+1} slots: Invariant 1 guarantees that there must be $2^{i+1} - (2^i \mod 2^{i+1}) = 2^i$ hot elements in the first 2^{i+1} slots. Partitioning precisely takes advantage of this fact: by partitioning, P moves the 2^i hottest elements into the first 2^i slots. Thus, by partitioning we ensure that *Invariant* 1 holds in time-step $t + 1$.

RAM is correct. □

The RAM provides 2 operations:

- `initRAM` takes as an argument the desired RAM size. It initializes an array of the specified size where each slot contains a pair of a constant authentic index and 0 (the element and its index are permuted together). I.e., the RAM is 0 initialized. The RAM maintains a persistent time-step t that is used to decide when to repartition. t is initialized to 0.
- `accessRAM` takes 4 arguments: (1) the RAM to read from/write to, (2) an authentic flag which indicates whether or not to write, (3) an authentic index, and (4) an authentic value to write. `accessRAM` first applies `rearrange` to move hot items closer to the front of RAM. Then, it reads the item in slot 0 and increments t. If the flag argument is set, then it overwrites the element in slot 0. `accessRAM` forces P to prove that her argument index is equal to the stored index (i.e. P includes the difference between the two indices in `zerosActual`) and returns the looked up element.

We prove the following theorem in Appendix B:

Theorem 6.4 (`BubbleRAM` Communication). `BubbleRAM` *with n elements incurs $\frac{1}{2}\log^2 n$ amortized OTs per call to* `accessRAM`.

We emphasize the high concrete efficiency of `BubbleRAM`: at only 4 elements, `BubbleRAM` already exceeds the performance of a linear scan RAM. Linear scans incur costly index comparisons, while `BubbleRAM` does not. Instead, P provides a single MAC confirming that she placed the correct element in front.

Informally, it is easy to see that `BubbleRAM` is secure, since the entire RAM is implemented as a circuit. We do allow P to *freely* partition RAM, but we (1) store each element alongside its index, (2) partition elements and indexes together, and (3) check that the accessed index matches the stored index. This ensures the integrity of RAM operations. We elaborate on security in Appendix A.

7 SECURITY

ZKM is a *secure* verifiable projective garbling scheme [4, 18, 27]. Specifically, it satisfies the following required properties: ZKM is **correct**, **sound**, and **verifiable**. In this section we formally present these properties and state the relevant theorems. For a lack of space, we defer the proofs of these theorems to Appendix A.

Definition 7.1 (Correctness). A garbling scheme is **correct** if for all programs $p \in \texttt{program}$ and all inputs $i \in \texttt{inp}$ where $\texttt{ev}(p, i) = 1$:

$$(e, d) = \texttt{Gb}(1^\sigma, p) \implies \texttt{Ev}(p, \texttt{En}(e, i), i) = d$$

Correctness formally states that our approach implements the processor described in Section 6.1.

Theorem 7.2 (Correctness). *If the prime modulus q is greater than $(2^{32} - 1)^2$, then* ZKM *is **correct**.*

Definition 7.3 (Soundness). A garbling scheme is **sound** if for all programs $p \in \texttt{program}$, all inputs $i \in \texttt{inp}$ such that $\texttt{ev}(p, i) = 0$, and all probabilistic polynomial time adversaries $\mathcal{A}$ the following probability is negligible in σ:

$$Pr(\mathcal{A}(p, \texttt{En}(e, i)) = d : (e, d) \leftarrow \texttt{Gb}(1^\sigma, p))$$

Soundness ensures that a cheating prover cannot win: a prover who does not have a witness cannot construct the secret d.

Theorem 7.4 (Soundness). ZKM *is **sound**.*

Definition 7.5 (Verifiability). A garbling scheme is **verifiable** if for all programs $p \in \texttt{program}$, all $i \in \texttt{inp}$ such that $\texttt{ev}(p, i) = 1$, and all probabilistic polynomial time adversaries $\mathcal{A}$ there exists an expected polynomial time algorithm Ext such that the following probability is negligible in σ:

$$Pr(\texttt{Ext}(p, e) \neq \texttt{Ev}(p, \texttt{En}(e, i)) : (e, \cdot) \leftarrow \mathcal{A}(1^\sigma, p), \texttt{Ve}(p, e) = 1)$$

Verifiability ensures that the garbling scheme supports malicious verifier Zero Knowledge. The prover P uses the procedure Ve to verify that V did not cheat when constructing the encoding e. Additionally, the property ensures that V learns nothing from running the protocol, since he knows the secret d ahead of time: Ext extracts d from e in polynomial time.

Theorem 7.6 (Verifiability). ZKM *is **verifiable**.*

8 INSTANTIATION

We implemented our ZK processor in 1900 lines of C++. We set the prime modulus q to $2^{64} - 59$, the largest 64 bit prime. Hence, our statistical security parameter σ is 64. We choose this q for two reasons: (1) it is greater than $(2^{32} - 1)^2$ and hence satisfies the correctness requirement of our construction and (2) MACs fit into 64 bit integers, yielding high computational performance.

We instantiate OT with the state-of-the-art malicious OT extension protocol of [29] using the implementation provided by [40]. Thus, each OT communicates 48 bytes: 16 to send a random OT and 32 to transfer both secret pairs.

Optimizations. $\widehat{\texttt{step}}$ (Figure 2) focuses on clarity. However, it is also somewhat inefficient. For example, the specification repeatedly compares values to *op*. Each performed comparison requires a bit decomposition, and therefore incurs many OTs. Appendix D lists some small but critical optimizations that improve performance.

Component	Cost (OTs)
Decode instruction	86
Registry RAM read (amortized)	26
Read P input	32
Multiplication/project arg_0	32
Bitwise multiplication/project arg_1	32
Comparisons	65
Misc. Boolean multiplications	25
Registry RAM write (amortized)	14
Mod 2^{32}	64
Total (amortized)	376

Figure 6: The OT costs per cycle of processor components.

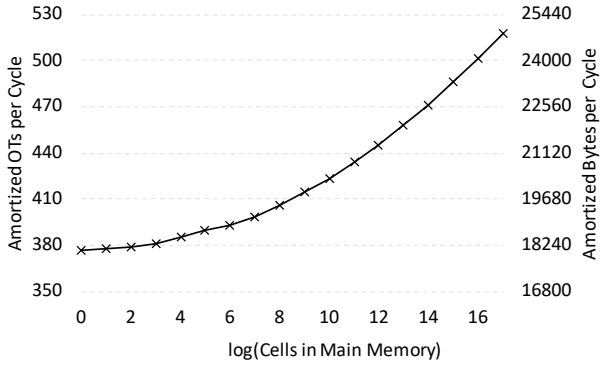

Figure 7: Total amortized OTs and corresponding communication cost per cycle as a function of main memory size. Each proof incurs a one-time communication cost of 150 KB due to base OTs.

Benchmark hardware and network. In Section 9, we evaluate our approach with benchmarks. 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. All experiments were run on a simulated 1Gbps LAN with 2ms latency.

9 EVALUATION

Subcomponent cycle costs. We exercised the ZKM machinery on a simple program computing the factorial of P's private input n and comparing the output to a fixed value. We ran ZKM for 32 cycles to fully exercise partitioning of the registry RAM (see Section 6.8) and amortized the OTs across cycles. Figure 6 tabulates the cost of the various components in the optimized $\widehat{\texttt{step}}$ algorithm, ignoring main memory. This table shows that our ZK RAM pays dividends: register accesses are among the cheapest operations in each cycle. It also highlights the costly parts of our system: decoding instructions, comparison, and computing mod 2^{32} stand out. The total cycle cost is 376 OTs, or around 17.6KB of communication, per cycle.

We next analyze the efficiency of BubbleRAM. For this experiment, we ran ZKM on the same factorial benchmark, but varied the size of main memory (recall that RAM is accessed every cycle). We ran ZKM for enough cycles to fully exercise the partitioning of main memory (i.e., for RAM size n we ran n cycles). Figure 7 shows the amortized OT and communication cost per cycle as a function of main memory size. The results emphasize the excellent concrete efficiency of BubbleRAM: even with 2^{17} memory cells (i.e. a 512KB RAM), the memory access cost was less than half of the cycle cost.

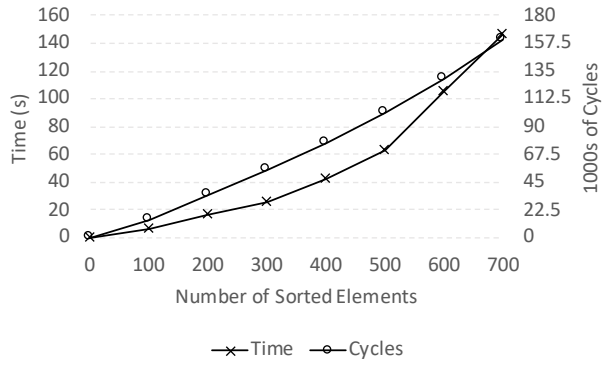

Figure 8: Time and cycle cost of sorting random integers, then proving the existence of an invalid memory dereference. Wall-clock time is the end-to-end time for P to expand her input (Section 6.4) and then for V and P to run the protocol. In each experiment, the main memory has 2^{12} cells (i.e. 16KB). Measurements were averaged over 5 runs and fresh random arrays were chosen for each run.

ZK bugs benchmark. [24] presents an exciting use-case: ZKP that a program contains a bug. More precisely, P proves knowledge of an input that causes V's program to trigger a specific behavior, for example accessing an array out of bounds. While [24] efficiently handles limited conditional branching, our architecture handles fully general control flow. We evaluate our performance on the example benchmark in [24], which is a snippet of C code (cf. Appendix G) that dereferences unallocated memory on specific inputs. We adapted this snippet to our instruction set. The full proof takes 0.42s, slower than the 0.1s of [24]. However, our approach is significantly more general. For instance, the snippet contains usages of the C standard library functions `strlen` and `strncmp`. Both of these functions iterate over C character strings. However, because [24] encodes the circuit directly, they explicitly bound these loops to a fixed number of iterations. Our versions of these C standard functions are not bounded and hence more precisely model their behavior.

Our approach has another significant advantage over [24]: our efficient ZK RAM. This is critical as typical programs access RAM frequently. We explore this with a new benchmark (code is in Appendix G) that aggressively uses RAM: we modify [24] code to first sort an array of random values using the `quick_sort` algorithm[4]. After sorting, P provides an input triggering the invalid dereference.

Our sorting benchmark (and others with unstructured memory access) are extremely expensive to directly encode as circuits: the `quick_sort` circuit has cubic size because (1) we must be maximally pessimistic about partitioning and (2) due to linear-scan RAM accesses. Back-of-the-napkin math suggests that to sort an array of 500 elements, [24] requires ≈ 500GB of communication. Further, an expert must annotate each loop with an explicit upper bound. In contrast, while ZKM individual cycles are relatively costly, we can represent programs like `quick_sort` *far* more efficiently and without expert annotations.

[4]It is possible to substitute `quick_sort` by an efficient oblivious sorting algorithm, but the purpose of this benchmark is to show how ZKM scales with RAM-heavy programs, not specifically to sort arrays.

n	BubbleRAM	**Floram** [17]	**Lookahead RAM** [35]
2^5	0.61	~ 600 / 980×	~ 5 / 8×
2^7	1.05	~ 610 / 580×	~ 25 / 23×
2^9	1.77	~ 620 / 350×	~ 50 / 28×
2^{11}	2.69	~ 640 / 240×	~ 90 / 33×
2^{13}	3.82	~ 670 / 175×	~ 160 / 42×
2^{15}	5.13	~ 700 / 135×	no data provided
2^{17}	6.63	~ 730 / 110×	no data provided

Figure 9: Comparing BubbleRAM to state-of-the-art ORAMs. n is the number of elements in RAM. We tabulate communication per access in KB. Communication factor improvement over related work is given. Related work performance is approximated from plots respectively in [17] and [35].

Figure 8 shows the efficiency of our approach as a function of the array size. Peak performance occurs in the 100, 200, and 300 element instances. In these benchmarks, ZKM achieves 2.1KHz. For 500 elements, our proof uses 101K cycles and 2GB of communication.

In Figure 8, wall-clock time grows faster than the numbers of cycles. This discrepancy is caused by computational overhead associated with storing large numbers of OT inputs/outputs. We believe future work will improve the computation of our approach.

Comparing BubbleRAM *to existing ORAM.* To our knowledge, BubbleRAM is the first ZK-specific ORAM construction. Prior ZK works interface with ORAM in a black-box manner [25, 33]. Thus, we compare BubbleRAM to existing concretely efficient ORAMs. Of course, while ours turns out to be significantly cheaper than existing ORAM, ours is specific to the ZK setting.

Floram is the state-of-the-art in concrete efficiency for large ORAMs [17]. Floram outperforms Circuit ORAM and a square-root ORAM for RAMs with more than 2^{11} entries [17, 39, 43]. For smaller sizes, Lookahead ORAM, a recent square-root technique, is preferable [35]. We compare the communication cost of BubbleRAM to these two works in Figure 9. We tabulate [17]'s computation-expensive but communication-cheap 'CPRG Floram'. The results show that BubbleRAM outperforms these works in communication by large factors. This large improvement is possible because BubbleRAM is designed with ZK in mind. BubbleRAM also performs all accesses in constant rounds. Compared works require rounds of communication to access content.

Comparing ours to existing ZK processors. A persistent line of work has constructed succinct non-interactive ZK proofs in a processor model similar to ours [7, 10, 11]. Because of non-interactivity and succinctness, these works are applicable to more problems than our approach. In exchange, our approach is vastly more efficient. These approaches attain a clock rate less than 10Hz on powerful hardware and manipulate memories containing hundreds of bits. Ours runs at 2.1KHz on commodity hardware and can manipulate a main memory holding hundreds of KBs of data.

Acknowledgement. This work was supported in part by NSF award #1909769 and in part 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.

REFERENCES

[1] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. 2017. Ligero: Lightweight Sublinear Arguments Without a Trusted Setup. In *ACM CCS 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, 2087–2104. https://doi.org/10.1145/3133956.3134104

[2] Marshall Ball, Tal Malkin, and Mike Rosulek. 2016. Garbling Gadgets for Boolean and Arithmetic Circuits. In *ACM CCS 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, 565–577. https://doi.org/10.1145/2976749.2978410

[3] Mihir Bellare and Oded Goldreich. 1993. On Defining Proofs of Knowledge. In *CRYPTO'92 (LNCS)*, Ernest F. Brickell (Ed.), Vol. 740. Springer, Heidelberg, 390–420. https://doi.org/10.1007/3-540-48071-4_28

[4] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. 2012. Foundations of garbled circuits. In *ACM CCS 2012*, Ting Yu, George Danezis, and Virgil D. Gligor (Eds.). ACM Press, 784–796. https://doi.org/10.1145/2382196.2382279

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

[6] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2019. Scalable Zero Knowledge with No Trusted Setup. In *CRYPTO 2019, Part III (LNCS)*, Alexandra Boldyreva and Daniele Micciancio (Eds.), Vol. 11694. Springer, Heidelberg, 701–732. https://doi.org/10.1007/978-3-030-26954-8_23

[7] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In *CRYPTO 2013, Part II (LNCS)*, Ran Canetti and Juan A. Garay (Eds.), Vol. 8043. Springer, Heidelberg, 90–108. https://doi.org/10.1007/978-3-642-40084-1_6

[8] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. 2019. Aurora: Transparent Succinct Arguments for R1CS. In *EUROCRYPT 2019, Part I (LNCS)*, Yuval Ishai and Vincent Rijmen (Eds.), Vol. 11476. Springer, Heidelberg, 103–128. https://doi.org/10.1007/978-3-030-17653-2_4

[9] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2013. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. Cryptology ePrint Archive, Report 2013/879. (2013). https://eprint.iacr.org/2013/879.

[10] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Scalable Zero Knowledge via Cycles of Elliptic Curves. In *CRYPTO 2014, Part II (LNCS)*, Juan A. Garay and Rosario Gennaro (Eds.), Vol. 8617. Springer, Heidelberg, 276–294. https://doi.org/10.1007/978-3-662-44381-1_16

[11] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In *USENIX Security 2014*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 781–796.

[12] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. 2018. Bulletproofs: Short Proofs for Confidential Transactions and More. In *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 315–334. https://doi.org/10.1109/SP.2018.00020

[13] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. 2017. Post-Quantum Zero-Knowledge and Signatures from Symmetric-Key Primitives. In *ACM CCS 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, 1825–1842. https://doi.org/10.1145/3133956.3133997

[14] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. 2015. Geppetto: Versatile Verifiable Computation. In *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 253–270. https://doi.org/10.1109/SP.2015.23

[15] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. 1994. Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols. In *CRYPTO'94 (LNCS)*, Yvo Desmedt (Ed.), Vol. 839. Springer, Heidelberg, 174–187. https://doi.org/10.1007/3-540-48658-5_19

[16] Alfredo De Santis and Giuseppe Persiano. 1992. Zero-Knowledge Proofs of Knowledge Without Interaction (Extended Abstract). In *33rd FOCS*. IEEE Computer Society Press, 427–436. https://doi.org/10.1109/SFCS.1992.267809

[17] Jack Doerner and abhi shelat. 2017. Scaling ORAM for Secure Computation. In *ACM CCS 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, 523–535. https://doi.org/10.1145/3133956.3133967

[18] Tore Kasper Frederiksen, Jesper Buus Nielsen, and Claudio Orlandi. 2015. Privacy-Free Garbled Circuits with Applications to Efficient Zero-Knowledge. In *EUROCRYPT 2015, Part II (LNCS)*, Elisabeth Oswald and Marc Fischlin (Eds.), Vol. 9057. Springer, Heidelberg, 191–219. https://doi.org/10.1007/978-3-662-46803-6_7

[19] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. 2013. Quadratic Span Programs and Succinct NIZKs without PCPs. In *EUROCRYPT 2013 (LNCS)*, Thomas Johansson and Phong Q. Nguyen (Eds.), Vol. 7881. Springer, Heidelberg, 626–645. https://doi.org/10.1007/978-3-642-38348-9_37

[20] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. 2016. ZKBoo: Faster Zero-Knowledge for Boolean Circuits. In *USENIX Security 2016*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 1069–1083.

[21] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1991. Proofs That Yield Nothing but Their Validity or All Languages in NP Have Zero-knowledge Proof Systems. *J. ACM* 38, 3 (July 1991), 690–728. https://doi.org/10.1145/116825.116852

[22] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. 1985. The Knowledge Complexity of Interactive Proof-Systems (Extended Abstract). In *17th ACM STOC*. ACM Press, 291–304. https://doi.org/10.1145/22145.22178

[23] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *EUROCRYPT 2016, Part II (LNCS)*, Marc Fischlin and Jean-Sébastien Coron (Eds.), Vol. 9666. Springer, Heidelberg, 305–326. https://doi.org/10.1007/978-3-662-49896-5_11

[24] David Heath and Vladimir Kolesnikov. 2020. Stacked Garbling for Disjunctive Zero-Knowledge Proofs. Cryptology ePrint Archive, Report 2020/136. (2020). https://eprint.iacr.org/2020/136.

[25] Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. 2015. Efficient Zero-Knowledge Proofs of Non-algebraic Statements with Sublinear Amortized Cost. In *CRYPTO 2015, Part II (LNCS)*, Rosario Gennaro and Matthew J. B. Robshaw (Eds.), Vol. 9216. Springer, Heidelberg, 150–169. https://doi.org/10.1007/978-3-662-48000-7_8

[26] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. 2007. Zero-knowledge from secure multiparty computation. In *39th ACM STOC*, David S. Johnson and Uriel Feige (Eds.). ACM Press, 21–30. https://doi.org/10.1145/1250790.1250794

[27] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. 2013. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *ACM CCS 2013*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM Press, 955–966. https://doi.org/10.1145/2508859.2516662

[28] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. 2018. Improved Non-Interactive Zero Knowledge with Applications to Post-Quantum Signatures. In *ACM CCS 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, 525–537. https://doi.org/10.1145/3243734.3243805

[29] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2015. Actively Secure OT Extension with Optimal Overhead. In *CRYPTO 2015, Part I (LNCS)*, Rosario Gennaro and Matthew J. B. Robshaw (Eds.), Vol. 9215. Springer, Heidelberg, 724–741. https://doi.org/10.1007/978-3-662-47989-6_35

[30] Vladimir Kolesnikov and Thomas Schneider. 2008. Improved Garbled Circuit: Free XOR Gates and Applications. In *ICALP 2008, Part II (LNCS)*, Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz (Eds.), Vol. 5126. Springer, Heidelberg, 486–498. https://doi.org/10.1007/978-3-540-70583-3_40

[31] Vladimir Kolesnikov and Thomas Schneider. 2008. A Practical Universal Circuit Construction and Secure Evaluation of Private Functions. In *FC 2008 (LNCS)*, Gene Tsudik (Ed.), Vol. 5143. Springer, Heidelberg, 83–97.

[32] Steve Lu and Rafail Ostrovsky. 2013. How to Garble RAM Programs. In *EUROCRYPT 2013 (LNCS)*, Thomas Johansson and Phong Q. Nguyen (Eds.), Vol. 7881. Springer, Heidelberg, 719–734. https://doi.org/10.1007/978-3-642-38348-9_42

[33] Payman Mohassel, Mike Rosulek, and Alessandra Scafuro. 2017. Sublinear Zero-Knowledge Arguments for RAM Programs. In *EUROCRYPT 2017, Part I (LNCS)*, Jean-Sébastien Coron and Jesper Buus Nielsen (Eds.), Vol. 10210. Springer, Heidelberg, 501–531. https://doi.org/10.1007/978-3-319-56620-7_18

[34] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly Practical Verifiable Computation. In *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 238–252. https://doi.org/10.1109/SP.2013.47

[35] Michael Raskin and Mark Simkin. 2019. Perfectly Secure Oblivious RAM with Sublinear Bandwidth Overhead. 11922 (2019).

[36] Claus-Peter Schnorr. 1990. Efficient Identification and Signatures for Smart Cards. In *CRYPTO'89 (LNCS)*, Gilles Brassard (Ed.), Vol. 435. Springer, Heidelberg, 239–252. https://doi.org/10.1007/0-387-34805-0_22

[37] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. 2015. TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits. In *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 411–428. https://doi.org/10.1109/SP.2015.32

[38] Abraham Waksman. 1968. A Permutation Network. *J. ACM* 15, 1 (Jan. 1968), 159âĂŞ163. https://doi.org/10.1145/321439.321449

[39] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. 2015. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *ACM CCS 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM Press, 850–861. https://doi.org/10.1145/2810103.2813634

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

[41] Xiao Shaun Wang, S. Dov Gordon, Allen McIntosh, and Jonathan Katz. 2016. Secure Computation of MIPS Machine Code. In *ESORICS 2016, Part II (LNCS)*, Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows (Eds.), Vol. 9879. Springer, Heidelberg, 99–117. https://doi.org/10.1007/978-3-319-45741-3_6

[42] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. 2019. Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation. In *CRYPTO 2019, Part III (LNCS)*, Alexandra Boldyreva and Daniele Micciancio (Eds.), Vol. 11694. Springer, Heidelberg, 733–764. https://doi.org/10.1007/978-3-030-26954-8_24

[43] Samee Zahur, Xiao Shaun Wang, Mariana Raykova, Adria Gascón, Jack Doerner, David Evans, and Jonathan Katz. 2016. Revisiting Square-Root ORAM: Efficient Random Access in Multi-party Computation. In *2016 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 218–234. https://doi.org/10.1109/SP.2016.21

A SECURITY - EXTENDED

This section extends the abbreviated discussion of security in Section 7. We prove that ZKM is a *secure* verifiable projective garbling scheme [4, 18, 27]. Specifically, ZKM is **correct**, **sound**, and **verifiable**.

We now prove Theorem 7.2.

Proof. By induction on the processor state over steps and the correctness of the individual components used in $\widehat{\texttt{step}}$.

The correctness of $\widehat{\texttt{step}}$ relies on the correctness of the following:

- The homomorphic value operations of addition, subtraction, and multiplication by a public constant. An algebraic argument for the correctness of addition is given in Section 6.5. The other two operations can be proven correct similarly.
- The non-homomorphic value operations of multiplication, bit decomposition, and mod 2^{32}. Arguments for the correctness of each of these is given in Section 6.6.
- Value comparison operations (e.g. ==, <). These operations are implemented by first performing bit decomposition on both inputs, then implementing the comparisons via Boolean operations. We explain the translation of Boolean operations to arithmetic operations in Section 6.6.
- The procedure readInput. Correctness of this procedure is argued in Section 6.6.
- The ROM. Correctness of the ROM is argued in Section 6.7.
- The RAM. Correctness of the RAM is argued in Section 6.8.

Now, since the inputs to $\widehat{\texttt{step}}$ are assumed correct by induction and each component used in $\widehat{\texttt{step}}$ is correct, we need only argue that $\widehat{\texttt{step}}$ (Figure 2) accurately carries out the behavior in step (Figure 1). By inspecting the two procedures, two key differences stand out:

(1) step conditionally dispatches over the instruction whereas $\widehat{\texttt{step}}$ does not. However, $\widehat{\texttt{step}}$ uses algebra to encode the same conditional behavior.
(2) step manipulates 32 bit values directly, and hence all intermediate values are 32 bit values. In contrast, $\widehat{\texttt{step}}$ manipulates values in $\mathbb{Z}_q$. Note that before writing values into the registry, $\widehat{\texttt{step}}$ first computes mod32. Thus, all authentic values in the registry and memory encode 32 bit integers. It remains only to show that no intermediate values 'overflow' q; an overflow would lose precision and compromise correctness. However, the only operations which can escape this range are add and mul, of which multiplication can overflow more. However, given that both inputs to multiplication are 32 bit values, the highest possible product is $(2^{32}-1)^2$. But we assumed q is greater than this maximum product, and so an overflow is impossible and hence multiplication is correct.

ZKM is correct. □

We now prove Theorem 7.4.

Proof. By the security of the arithmetic representation and induction on the overall circuit.

Recall that the key property of the arithmetic representation is that the two players maintain shares of MACs for each wire. In particular for a wire with value a, the two players maintain the following shares:

$$\langle A, a \cdot \Delta - A \rangle$$

where $A, \Delta \leftarrow_{\$} \mathbb{Z}_q$ and q is a σ-bit prime. Since A, Δ are uniform and q is prime, the share $a \cdot \Delta - A$ is uniform. Suppose that P wishes to cheat and forge a different value $a' \neq a$ on the same wire. Then P must find the value $a' \cdot \Delta - A$. But the value A is chosen independently for each wire and hence is independent of all other messages received. Furthermore, the value Δ is always masked. Thus, P can correctly select a value $a' \cdot \Delta - A$ with probability $\frac{1}{q}$, and therefore has negligible probability of choosing such a value.

Next, we show that low level operations preserve the authenticity of MACs by induction. For example, when the parties compute mod32 we ensure that the output MAC depends on the input MAC (which is authentic by induction). In this and other operations (multiplication, projecting to bitwise representation), P is forced to demonstrate that her choices add together to a MAC equal to the input MAC by appending to zerosActual. Thus, the output MAC is determined completely by the input MACs and hence the output MAC is authentic.

RAM and ROM operations are an exception to this: we allow P to arbitrarily permute memory without checking authenticity of the permutation. However, when memory is accessed, we check that the given index (which is authentic by induction) is equal to the index stored in the memory.

Thus, the resulting read/write is authentic. ZKM is sound. □

We now prove Theorem 7.6.

Proof. By constructing a polynomial time algorithm Ext.

First, note that e stores all randomness drawn by V: for all processor wires whose value is determined by OT, e holds both P's share of a 0 MAC and of a 1 MAC. 0 MACs are randomly drawn, and furthermore, constitute *all* of the random choices made by V. Thus, Ext can use e to extract all of V's randomness. Now, Ext simply runs the processor in VERIFIER mode, except that when a procedure indicates to draw randomness, Ext instead uses the corresponding randomness in e. Finally, Ext outputs $d = H(\texttt{zerosExpected})$. At a high level, the above shows that d is based only on V's randomness, and thus V learns nothing when P sends the proof.

Now, we show that even if V is adversarial, this extraction still succeeds. This is based on the correctness of ZKM.Ve. ZKM.Ve allows P to check that e is consistent with the protocol. In our case, ZKM.Ve is nearly identical to the extraction algorithm: It runs the processor in CHECK mode, using the randomness in e. As it runs, it checks each entry in e by ensuring the difference between the 0 choice and the 1 choice is consistent with the processor operations, e.g. checking that the OTs are all consistent with a single global Δ.

Note that regardless of the value of e, if ZKM.Ve outputs 1, then Ext will output a value d consistent with evaluation. This is because both algorithms are based on running the processor using the randomness in e.

ZKM is verifiable. □

B BubbleRAM CONCRETE EFFICIENCY

Section 6.8 deferred the proof of concrete communication cost. We prove Theorem 6.4:

Proof. By amortizing the costs of partitions to accesses.

Consider a partition of size $2i$; i.e., a partition that places the next i elements at the front of RAM. This partition costs $i \log i + \frac{i}{2}$ OTs. This cost can be amortized across the next i accesses: the partition has no goal other than to place the next i elements at the front of RAM. Thus for this partition, each of the next i accesses incurs amortized cost $\log i + \frac{1}{2}$.

Now, consider a single element as it moves closer to the front of the array in order to be accessed. To ensure that this element ends up in slot 0 such that it can be accessed, the ORAM uses $\log n$ partitions: a partition of size n to place it in the first $\frac{n}{2}$ elements, a partition of size $\frac{n}{2}$ to place it in the first $\frac{n}{4}$ elements, …, and a partition of size 2 to place it in index 0. That is, the total amortized OT cost is expressed by the following summation:

$$\sum_{i=0}^{\log n - 1} \left(\log(2^i) + \frac{1}{2} \right) = \frac{1}{2} \log^2 n$$

□

C $2 \log n$ ROM

In this section we explain the amortized $2 \log n$ ROM that implements our program texts. At a high level, the ROM uses the fact that P knows ahead of time the order in which instructions will be accessed. We use this to plan ahead many instructions and to amortize costly reordering algorithms.

In more detail, we allow P to choose an arbitrary permutation of the ROM such that instructions appear in their access order. Two problems remain: (1) we are not yet checking that P's permutation is correct and (2) the next n instructions to be executed might contain *repeated* instructions (i.e. a given instruction might be needed more than once). We solve these problems as follows:

(1) Instead of storing just elements in ROM, we store each element *paired with its index*. That is, we explicitly store an index value $i \in \hat{v}$ in the ROM alongside each element. During permutation, we permute both the element and its index together. Then, to read the ROM, we check that the provided index matches the stored index. P must include the difference between these two indexes as part of her proof. Since the provided index is authentically computed, the result of reading the ROM is also authentic.

(2) We account for repeated instructions by using a construction in [31]: namely a *selection* block. A selection block is an oblivious construction that allows the 'permutation with copies' that we need. It is built from 4 steps: (1) Determine which of the n elements are needed and which are unneeded in the next n accesses. (2) Permute the ROM such that each needed element that requires i copies is followed by $i - 1$ unneeded elements. (3) Linearly scan the ROM, replacing unneeded elements by the preceding element based on a bit provided by P. Thus, the ROM now contains sufficient copies of each needed element. Note that conditional replacement can be achieved with a swap gate, and hence conditionally replacing a 32 bit value requires a single OT. (4) Permute the copies into the access order.
We apply a selection block to the ROM content every n reads. Thus, to prepare for n reads, the ROM requires 2 permutations and 1 linear scan (which costs n OTs).

Overall, n read-only memory accesses cost $2n \log n$ OTs and thus each access costs amortized $2 \log n$ OTs.

The operations on the ROM are defined as follows:

- initROM takes as an argument a cleartext array of values. Then, it (1) pairs each element with that element's index, (2) converts both items in the pair to authentic values using our representation's support for encoding public constants, and (3) outputs the resulting array.
- readROM takes as an argument an authentic index. Internally, readROM maintains a persistent counter c that specifies which entry in the ROM to read next. c is initially set to the ROM's size (this causes the ROM to be rearranged on the first access). readROM checks if c is equal to the ROM's size. If so, then the ROM is rearranged using a selection block programmed by P's private input and c is reset to 0. Then, the cth index is read from the ROM array. Recall that this entry includes a pair of an authentic index and the element itself. readROM forces P to prove that the provided index argument is equal to this stored index (i.e. P includes the difference between the two indices in zerosActual), increments c, and returns the accessed element.

D IMPLEMENTATION OPTIMIZATIONS

Our core specification $\widehat{\texttt{step}}$ performs many redundant operations to maintain clarity. Here, we list some of the simple optimizations we use to improve overall performance:

- Instead of repeatedly comparing values to *op*, we instead implement a binary decoder circuit that efficiently generates a bit-map where exactly one bit in the output is 1. The 1 bit indicates which operation to perform, and thus prevents us from needing to repeatedly compare *op* against constants.
- We amortize low-level bit operations where possible. In our formalization, we present < as the only comparator, but in our implementation we perform all comparisons as well as other bitwise operations. Bit operations are highly redundant across each of these comparisons/bitwise operations, so we reuse them to reduce communication cost.
- We amortize the project operation with multiplication and with bitwise multiplication. Notice that mul32 and mul1 both *already* require us to perform a binary decomposition in order to check the validity of P's OT selections. Therefore, we can combine multiplication/bitwise multiplication with project to save OTs.

P sends x to $\mathcal{F}_{COT}$
$\mathcal{F}_{COT}$ sends chosen to V
V runs $(e, d) \leftarrow \mathsf{Gb}(1^\kappa, prog)$
V sends e to $\mathcal{F}_{COT}$
$\mathcal{F}_{COT}$ sends $\mathsf{En}(e, x)$ to P
P runs $Y \leftarrow \mathsf{Ev}(prog, X)$; if Ev aborts, $Y \leftarrow \perp$
P sends $(\mathtt{commit}, 1, Y)$ to $\mathcal{F}_{COM}$
$\mathcal{F}_{COM}$ sends $(\mathtt{committed}, 1, H(Y))$ to V
V sends open-all to $\mathcal{F}_{COT}$
$\mathcal{F}_{COT}$ sends $(\mathtt{transfer}, i, e)$ to V
P runs $b \leftarrow \mathsf{Ve}(prog, e)$
if $b \neq 1$ **then**
  P aborts
P sends reveal to $\mathcal{F}_{COM}$
$\mathcal{F}_{COM}$ sends Y to V
if $\mathsf{De}(Y, d)$ **then**
  V outputs accept

Figure 10: The protocol for garbling scheme based ZK. $\mathcal{F}_{COM}$ is the committing OT functionality.

E MALICIOUS VERIFIER ZK PROTOCOL

Our approach is formalized as a verifiable garbling scheme. Thus, we can use the malicious verifier Zero Knowledge protocol of [27] directly. For completeness, we provide their algorithm in Figure 10.

F ADDITIONAL ALGORITHMS

We formalize algorithms that were deferred from the main paper.

Figure 11 lists the algorithm for computing 32-bit multiplication. The algorithm is similar to mul1 (Figure 4) except that 32 OTs are involved and 32-bit values are reconstructed via homomorphic addition and constant multiplication.

Figure 12 lists the algorithm for projecting an n-bit value into n 1-bit values. The algorithm is similar in flavor to multiplication algorithms: P simply asks for the correct projection, then proves that the projection matches the input by applying homomorphic operations to the projection.

Figure 13 provides the algorithm for reading P's private input. Here, P asks for a bitwise representation of her input via OT. Note that if $flag$ is 0 (indicating the current processor cycle does not need P's input), then the INPUT code does not pop the top of P's input stack. Instead, it 'waits' and asks for a MAC of 0 via OTs.

Figure 14 depicts a helper procedure, inject, which reconstructs a collection of n 1-bit values into a single n-bit value using homomorphic operations.

Figure 15 explains how to compute $\mathrm{mod} 2^{32}$ in our arithmetic representation. The algorithm is based off of our project procedure and simple arithmetic.

Figure 16 lists ZKM.En and ZKM.De that respectively explain which MACs P receives based on her input bits and how V checks that the proof is valid. ZKM.En is a straightforward mapping of bits into pairs. ZKM.De is a simple comparison.

Figure 17 lists ZKM.Ev and ZKM.Ve, two of the key algorithms in ZKM. We defer these algorithms to the appendix because they are nearly identical to ZKM.Gb. Both set the global mode, set some global variables, and defer to $\widehat{\mathsf{step}}$.

```
mul32([[a]], [[b]]) :
  ⟨a_V, a_P⟩ ← [[a]]
  ⟨b_V, b_P⟩ ← [[b]]
  switch(mode) :
    case(VERIFIER) :
      ab_V ← 0
      a'_V ← 0
      for i ∈ [0..32) :
        a'_i ←$ {0, q − 1}
        ab_i ←$ {0, q − 1}
        otInputs.push(
            (0 − a'_i, 0 − ab_i), (2^i · Δ − a'_i, 2^i · b_V − ab_i))
        ab_V ← ab_V + ab_i
        a'_V ← a'_V + a'_i
      zerosExpected.push(a_V − a'_V)
    case(INPUT) :
      for i ∈ [0..32)
        ▷ Choose the ith bit of a.
        otChoices.push((a&2^i) > 0)
    case(PROVER) :
      ab_P ← a · b_P
      a'_P ← 0
      for i ∈ [0..32)
        (a'_i, δ_i) ← otOutputs.pop()
        a'_P ← a'_P + a'_i
        ab_P ← ab_P + δ_i
      zerosActual.push(a'_P − a_P)
  return ⟨ab_V, ab_P⟩
```

Figure 11: The MAC multiplication procedure where the first argument $a \in \mathbb{Z}_{2^{32}}$.

We omit CHECK mode in the main body of our paper for brevity. For reference, Figure 18 includes the modified mul1 procedure that includes CHECK mode. Other procedures are similar.

```
project(n, [[a]]) :
  ⟨a_V, a_P⟩ ← [[a]]
  switch(mode) :
    case(VERIFIER) :
      a'_V ← 0
      for i ∈ [0..n) :
        V_i ←$ {0, q − 1}
        otInputs.push((0 − V_i, 0), (Δ − V_i, 0))
        a'_V ← a'_V + 2^i · V_i
      zerosExpected.push(a_V − a'_V)
    case(INPUT) :
      for i ∈ [0..n)
        ▷ Choose the ith bit of a.
        otChoices.push((a&2^i) > 0)
    case(PROVER) :
      a'_P ← 0
      for i ∈ [0..n)
        (P_i, ·) ← otOutputs.pop()
        a'_P ← a'_P + 2^i · P_i
      zerosActual.push(a'_P − a_P)
  ▷ Initialize an array of n bit MACs.
  out ← [⟨0, 0⟩ ; n]
  for i ∈ [0..n)
    out[i] ← ⟨V_i, P_i⟩
  return out
```

Figure 12: The projection procedure `project` which converts an authenticated value $a \in \mathbb{Z}_{2^l}$ into l authenticated values in $\{0, 1\}$. That is, `project` computes the authentic binary decomposition of the input into n bits.

```
readInput(flag) :
  switch(mode) :
    case(VERIFIER) :
      a_V ← 0
      for i ∈ [0..n) :
        a_i ←$ {0, q − 1}
        otInputs.push((0 − a_i, 0), (Δ − a_i, 0))
        a_V ← a_V + 2^i · a_i
    case(INPUT) :
      ▷ Convert P's 32 bit input into binary OT choices.
      if flag ≠ 0 : a ← I.pop()
      ▷ When flag is false, P fakes an additional input 0
      else : a ← 0
      for i ∈ [0..n)
        ▷ Choose the ith bit of a.
        otChoices.push((a&2^i) > 0)
    case(PROVER) :
      a_P ← 0
      for i ∈ [0..n)
        (a_i, ·) ← otOutputs.pop()
        a_P ← a_P + 2^i · a_i
  return mul1(flag, ⟨a_V, a_P⟩)
```

Figure 13: The procedure `readInput` allows P to choose her independent input values via OT.

```
inject(bits) :
  out ← 0
  for i ∈ [0..|bits|)
    out ← out + 2^i · bits[i]
  return out
```

Figure 14: The injection procedure `inject` which converts an array of authenticated values in $\{0, 1\}$ into a single value. That is, `inject` reconstructs a value from its binary decomposition. `inject` is the conceptual dual to `project`. Note that `inject` is computed using only homomorphic operations.

```
mod32(a) :
  ▷ Project a into its binary decomposition.
  bits ← project(64, a)
  ▷ Collect the high 32 bits into an array.
  hi ← bits[32..64]
  ▷ Subtract off the high bits.
  return a − (2^32 · inject(hi))
```

Figure 15: The `mod32` procedure computes $\text{mod}2^{32}$ on an authentic value. In practice, we use `mod32` to clamp values back into the cleartext value range. The most 'extreme' clamping is needed after a 32 bit multiplication: in this case, the argument a can encode a value as high as $(2^{32} - 1)^2$. Thus, `mod32` is designed for up to 64 bit values.

```
ZKM.En(otInputs, otChoices) :
  n ← |otChoices|
  ▷ Initialize an empty array of outputs.
  otOutputs ← [0 ; n]
  for i ∈ [0..n)|
    if otChoices[i]
    then otOutputs[i] ← otInputs[i].right
    else otOutputs[i] ← otInputs[i].left
  return otOutputs

ZKM.De(zerosExcepted, zerosActual) :
  return zerosExpected == zerosActual
```

Figure 16: ZKM.En and ZKM.De, the procedures that respectively explain how P's input bits are converted to MACs and how V checks the the proof object sent by P.

```
ZKM.Ev(prog, I) :
  ▷ set the global mode so that prover actions are taken.
  mode ← PROVER
  ▷ The input I contains all of P's OT outputs.
  otOutputs ← I
  (T, space, time) ← prog
  ▷ Initialize the processor state.
  s ← (initROM(T), 0, initRAM(32), initRAM(space))
  for i ∈ [0..time)
    step^(s)
  ▷ Read the output from register 0
  out ← accessRAM(s.registry^, 0, 0, 0)
  ▷ Check that the program output is 1.
  zerosExpected.push(1 − out)
  return H(zerosActual)

ZKM.Ve(prog, e) :
  ▷ set the global mode so that checking actions are taken.
  mode ← CHECK
  (T, space, time) ← prog
  ▷ Initialize the global encoding variable.
  ▷ Its contents are inspected in low level operations.
  encoding ← e
  ▷ Initialize the processor state.
  s ← (initROM(T), 0, initRAM(32), initRAM(space))
  for i ∈ [0..time)
    step^(s)
  ▷ Read the output from register 0
  out ← accessRAM(s.registry^, 0, 0, 0)
  ▷ If no low level actions failed, output 1.
  return 1
```

Figure 17: P's actions when running a program and when checking V's messages. Both procedures set relevant global variables and call step^ $time$ times.

```
mul1([[a]], [[b]]) :
  ⟨a_V, a_P⟩ ← [[a]]
  ⟨b_V, b_P⟩ ← [[b]]
  switch(mode) :
    case(VERIFIER) :
      a'_V ←$ {0, q − 1}
      ab_V ←$ {0, q − 1}
      otInputs.push((0 − a'_V, 0 − ab_V), (Δ − a'_V, b_V − ab_V))
      zerosExpected.push(a_V − a'_V)
    ...
    case(CHECK) :
      ((aV_0, δ_0), (aV_1, δ_1)) ← encoding.pop()
      if (Δ_guess == 0)
      ▷ P does not initially know Δ.
      ▷ Thus on the first OT that uses Δ, we must set up Δ
      ▷ to check consistency with other messages.
      then Δ_guess = aV_1 − aV_0
      else if (Δ_guess ≠ aV_1 − aV_0)
      then ABORT
      if (δ_1 − δ_0 ≠ b_V)
      then ABORT
      ab_V ← 0 − δ_0
  return ⟨ab_V, ab_P⟩
```

Figure 18: The modified `mul1` procedure that checks the correctness of V's messages.

```
int* partition(int* arr, int l, int h) {
  int x = arr[h];
  int i = l;
  int j = l;
  while (j < h) {
    if (arr[j] <= x) {
      int t = arr[i];
      arr[i] = arr[j];
      arr[j] = t;
      ++i;
    }
    ++j;
  }
  int t = arr[i];
  arr[i] = arr[h];
  arr[h] = t;
  return i;
}

void quick_sort(int* arr, int l, int h) {
  int size = h - l + 1;
  int* stack = malloc(size);
  int top = 0;
  stack[0] = l;
  stack[1] = h;
  top = 2;
  while (top != 0) {
    top--;
    h = stack[top];
    top--;
    l = stack[top];
    int p = partition(arr, l, h);
    if (p > l + 1) {
      stack[top++] = l;
      stack[top++] = p - 1;
    }
    if (p + 1 < h) {
      stack[top++] = p + 1;
      stack[top++] = h;
    }
  }
}
```

Figure 20: The `C` code for `quick_sort` that we adapted to our instruction set. Notice that the algorithm heavily depends on RAM usage by its arbitrary array accessing.

```
static const char* SMALL_BOARD = "small_board_v11";

int* alloc resources(const char* board_type) {
  int block_size;
  // The next line has a bug!!
  if (!strncmp(board_type, SMALL_BOARD,
              sizeof(SMALL_BOARD))) {
    block_size = 10;
  } else {
    block_size=100;
  }
  return malloc(block_size * sizeof(int));
}

int incr_clock(const char* board_type,
      int* resources) {
  int clock_loc;
  if (!strncmp(board type, SMALL_BOARD,
              strlen(SMALL_BOARD))) {
    clock_loc = 0;
  } else {
    clock_loc = 64
  }
  (*(resources + clock_loc))++;
  return resources[clock_loc];
}

void snippet(const char* board_type) {
  int* res = alloc_resources(board_type);
  incr_clock(board_type, res);
}
```

Figure 19: The `C` code snippet provided by [24]. This snippet contains a logic error that causes an invalid memory dereference when provided a specific input such as `"small_boaERROR"`. We can run this snippet in our ZK processor. P demonstrates there is a bug in 0.42s (Section 9).

G ZK BUGS BENCHMARK

We provide the `C` snippets which we adapted to our instruction set. Figure 19 lists the snippet from [24]. In this snippet, a malicious user can input particular values that cause an invalid memory dereference.

In our evaluation, we test our approach's main memory. To do so, we modify Figure 19 to first sort an array of values using `quick_sort`. Figure 20 lists the source code which we adapted to our instruction set. The snippet is interesting because it forces extensive use of random access memory: the inner `partition` algorithm has random array accesses which our approach excels at modeling.