EZEE
Epoch Parallel Zero Knowledge for ANSI C

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

Abstract

Recent work has produced interactive Zero Knowledge (ZK) proof systems that can express proofs as arbitrary C programs (Heath et al., 2021, henceforth referred to as ZEE); these programs can be executed by a simulated ZK processor that runs in the 10KHz range.

In this work, we demonstrate that such proof systems are amenable to high degrees of parallelism. Our *epoch parallelism*-based approach allows the prover and verifier to divide the ZK proof into pieces such that each piece can be executed on a different machine. These proof snippets can then be glued together, and the glued parallel proofs are equivalent to the original sequential proof.

We implemented and we experimentally evaluate an epoch parallel version of the ZEE proof system. By running the prover and verifier each across 31 2-core machines, we achieve a ZK processor that runs at up to 394KHz. This allowed us to run a benchmark involving the Linux program *bzip2*, which would have required at least 11 days with the former ZEE system, in only 8.5 hours.

1 Introduction

Zero knowledge (ZK) proofs (ZKPs) allow a prover $\mathcal{P}$ to demonstrate to a verifier $\mathcal{V}$ the truth of some statement, while revealing nothing additional. In particular, $\mathcal{P}$'s witness, which might be sensitive, remains hidden from $\mathcal{V}$. ZK is a powerful cryptographic primitive that enables numerous useful applications. As one simple example, prior work has shown that ZK can be used to allow $\mathcal{P}$ to prove to $\mathcal{V}$ the existence of a bug in a public program without leaking the source of the bug [HK20b].

For some time, cryptographers have known techniques for proving *arbitrary* statements in ZK. However, until relatively recently such statements needed to be encoded as Boolean or arithmetic *circuits*, and so it was difficult for non-experts to use this powerful technology. Moreover, naïve program-to-circuit unrolling is inefficient for many programs.

Recent work shows that it is practical to construct efficient ZK proof systems that operate over *RAM programs* rather than circuits [HK20a, HYDK21]. By choosing the RAM program to be a general purpose CPU and by implementing a compiler, it is now possible to encode arbitrary ZK proofs as ordinary ANSI-C programs [HYDK21]. These works present low-level *Zero-Knowledge machine* (ZKM) emulators, capable of running a complete instruction set in zero knowledge. Proof statements are input as C programs, compiled into the instructions of the ZKM, and then run on the ZKM. With these advances, implementing a ZK proof is as easy as writing a C program, practically opening ZK proofs to many new applications.

Despite advances in performance, state-of-the-art ZK processors run millions of times slower than commodity processors, executing instructions in only the low KHz range. Furthermore, given the inherent cryptographic overhead of ZK, it is unlikely that ZK machines will approach the performance of modern CPUs in the foreseeable future. A program that may modestly take a few seconds on a commodity processor may not complete in months when run in ZK. This high latency means that many ZK applications remain impractical.

In this work, we build a ZK system that greatly reduces proof latency by introducing a high degree of parallelism *without* needing to change the proof statement. Our *epoch parallelism* technique splits a logically sequential proof into different *epochs*. Each epoch, which can be thought of as a subsequence of instructions run during the program execution, can be handled by a pair of worker machines, one owned by $\mathcal{P}$ and one by $\mathcal{V}$. Because the epochs run in parallel, we decrease the proof latency by a factor up to the degree of available parallelism. The technique does incur a slightly larger proof, since $\mathcal{P}$ must additionally prove that the epochs are consistent, but this cost is low compared to the size of the overall proof.

1.1 Our Contributions

In this work we:

- Build on 'ZK for Everything and Everyone' (*ZEE*) [HYDK21] by designing, implementing, and evaluating *Epoch ZEE* (*EZEE*), a secure epoch parallel ZK proof system. We show that *EZEE* can execute off-the-shelf ANSI C programs inside ZK while utilizing epoch parallelism. We used *EZEE* to execute in ZK the Linux programs *sed* (proving it has a bug) and *bzip2* (from the industry standard SPEC2006 benchmark suite [Hen06], proving it terminates normally). In our experiments, we show that *EZEE* runs at up to 394KHz. This clockrate is bounded by the available parallelism, not by a limitation of the technique. With more processors, we estimate that *EZEE* can run at up to 1.8MHz (see Section 8).

- Provide a template that explains how ZK protocols can be transformed into epoch parallel ZK protocols. Formally, we specify an interface that we call the *PIM* (proof interface machine). We show that ZK machines that meet the *PIM* interface allow a general program transformation that introduces epoch parallelism. We believe *PIM* would also be useful in future ZKP parallelization work.

2 Related Work

Zero Knowledge Machines We present a ZK proof system in the RAM model of computation. This direction is relatively unexplored; we review the few works in the area.

The first such works built on succinct non-interactive ZK (NIZK) proof engines [BCTV14b, BCTV14a, BCG+13]. Although these works achieve highly desirable non-interactivity, they do not scale to machines powerful enough to handle large proofs. E.g., such machines only run in the 1Hz range. Building substantially more powerful NIZK machines remains an interesting research direction.

Recently, [HK20a] and subsequently [HYDK21] constructed far more efficient ZK RAM machines based on an *interactive* proof system. While both $\mathcal{P}$ and $\mathcal{V}$ must be online for the proof, the RAM machine runs thousands of times faster and can support a main memory with megabytes of RAM. Our work builds directly on the proof system of [HYDK21], 'ZK for Everything and Everyone', which we call *ZEE*, so we discuss the details of their system as background in Section 3.

By implementing epoch parallelism, we build a ZKP system with lower proof latency than the above systems.

[FKL+21] proposed a new efficient constant-overhead ZK RAM. It is concretely efficient and improves both in speed and supported RAM size over [HK20a] and [HYDK21]. While our epoch-parallel system builds on *ZEE* and BubbleRAM [HK20a], it should be possible to integrate their improved RAM into our epoch parallel approach. We leave this integration as future research; the focus of this work is exploring ZK parallelization.

Fast Interactive ZK Protocols A number of works investigate gate-by-gate interactive ZK protocols. Such works are interesting because they (1) achieve low proof latency and (2) can scale to large proof statements.

Our work builds on the information theoretic MAC (IT-MAC) based ZK proof system of [HK20a] and [HYDK21]. This proof system is based on the Garbled Circuit-based ZK paradigm (GC-ZK) initiated by [JKO13] and continued by [FNO15, KP17, HK20b]. We view GC-ZK and IT-MACs as background to our work (see Section 3). Note that we favor the protocol of [HYDK21] over the following discussed works because the authors provide a hand tuned CPU. E.g., they provide an ALU that was specifically designed with costs of the underlying ZK protocol in mind.

Mac'n'Cheese [BMRS20] is a recent ZK proof system that builds gate-by-gate interactive proofs on top of vector oblivious linear evaluation (VOLE). Their work also incorporates the recent stacked garbling technique [HK20b] to achieve efficient disjunctive proof statements.

Like [HK20a], Wolverine [WYKW21a] also builds on IT-MACs, but does so using a custom protocol rather than using the GC-ZK protocol of [JKO13]. Wolverine, which like Mac'n'Cheese is based on VOLE, is superceded by Quicksilver [YSWW20] (discussed shortly).

Line-Point ZK [DIO20] greatly simplified the handling of VOLE-based IT-MAC multiplication.

Quicksilver [YSWW20] combines Wolverine with Line-Point ZK to achieve an extremely communication-efficient ZK protocol. The authors argue that Quicksilver is communication optimal for the gate-by-gate paradigm, since each of their field multiplication gates requires only the transmission of one field element and one VOLE correlation (of course, approaches that do not operate gate-by-gate can achieve much lower communication, i.e. sublinear ZK).

Detailed comparison between Quicksilver and the protocol of [HYDK21] is not available. Nevertheless, Quicksilver [YSWW20] now appears to be the state-of-the-art protocol for gate-by-gate interactive ZK, particularly for low bandwidth networks. However, the system has not yet been applied to the RAM model of computation, so we favor the protocol of [HYDK21] which comes with a hand tuned CPU. Also, it is not clear that Quicksilver greatly outperforms the [HYDK21] protocol on fast networks, because the latter is based on OT instead of the more expensive VOLE. We view building a tuned CPU for Quicksilver and then applying epoch parallelism as important future work.

Non-interactive and Succinct ZK Many recent ZK works emphasize small proofs and/or non-interactivity, e.g. [GKR08,IKOS07,GMO16,CDG+17,AHIV17, KKW18,GGPR13,PHGR13,BCG+13,CFH+15,Gro16,BFH+20,BCR+19]. NIZK parallelization (of computation) has also been explored, e.g. [EFKP20,WZC+18]. While NIZK work has achieved very impressive results in terms of non-interactive proofs of smaller statements, such proof systems do not yet match the scale and low latency of the above interactive proof systems.

Epoch Parallelism Epoch parallelism is a technique that predicts future states of an execution and then uses those predicted states to parallelize that future execution. It has been used previously, typically for speculative acceleration [ZS02, NVCF08, VLW+12, SKW+10], in which the system predicts future behaviors of the system, and then speculatively runs that future execution using those predictions. These speculative executions are often run in parallel, in what is known as an *epoch-parallel* phase. If the predictions are correct, the system can remove bottlenecks such as I/O or heavy-weight computation. However, if the predictions are inaccurate, the system must roll-back and discard the work done during epoch-parallel execution.

Others have also used this technique with deterministic computation to help accelerate dynamic analyses [WDC+13, QDCF16]. Here the initial prediction step is on a deterministic computation, so the epoch generation is not speculative and will not roll-back. However, the predictor to generate the epoch's state is less expensive than the epoch execution, allowing for parallelization of the analysis code.

Our work demonstrates the natural compatibility of ZK proofs and epoch parallelism. By running the proof locally, $\mathcal{P}$ can easily predict with perfect precision future program states. Then, the slow-running portion of the ZK proof can be parallelized to a very high degree.

3 Preliminaries

Traditionally, cryptographers encoded ZK proofs as Boolean or arithmetic circuits. While circuits are theoretically convenient and are suitable for small proofs, it is difficult to express complex systems as simple circuits. Recent work shows that the state-of-the-art in ZK now suffices to support efficient CPU-emulation based proof systems [HK20a, HYDK21]. We build our epoch parallelism system on one such recent work, that allows ZK proofs to be encoded as ANSI C programs [HYDK21].

Thus, we briefly review their system and the cryptographic foundations on which it lies. From here on, we refer to this base system as *ZEE*.

3.1 Garbled Circuit Based ZK

[JKO13] were the first to achieve practical ZK proofs of arbitrary statements. The [JKO13] protocol builds efficient ZK on top of a simple semi-honest garbled circuit (GC) protocol. Here, $\mathcal{V}$ instantiates the GC generator and constructs a garbling of the proof statement encoded as a Boolean circuit. $\mathcal{V}$ sends this garbling to $\mathcal{P}$. Additionally, $\mathcal{V}$ conveys to $\mathcal{P}$ via oblivious transfer (OT) GC input labels that together encode $\mathcal{P}$'s witness. $\mathcal{P}$ then evaluates the garbled circuit gate by gate under encryption until finally obtaining a single output label; if this label encodes a logical one, then the proof succeeds. The *authenticity* property of GC ensures that even a malicious $\mathcal{P}$ cannot forge a convincing output label unless she has a valid witness. Thus, this technique elegantly and straightforwardly ensures that $\mathcal{P}$ cannot forge a proof.

Protecting against a malicious $\mathcal{V}$ is harder: $\mathcal{V}$ can, in particular, send an ill constructed circuit garbling that leaks part of $\mathcal{P}$'s witness and violates ZK security. [JKO13] guard against this by adding a simple commitment step: once $\mathcal{P}$ computes her GC output label, she does not directly send it to $\mathcal{V}$, but rather *commits* to it. Then, $\mathcal{V}$ sends to $\mathcal{P}$ a single PRG seed that was used to derive all garbling randomness. This seed allows $\mathcal{P}$ to replay $\mathcal{V}$'s actions when garbling the circuit and to check that all messages from $\mathcal{V}$ were properly constructed. Only once this check succeeds does $\mathcal{P}$ open her commitment.

While [JKO13] were the first to achieve efficient and arbitrary ZK, a cascade of research produced new ZK techniques, particularly in the space of succinct non-interactive ZK. Even so, the GC-ZK paradigm remains interesting because of its attractive performance characteristics: its communication and computation for both $\mathcal{P}$ and $\mathcal{V}$ scale linearly in the proof statement size with low constants. Moreover, thanks to OT extension [IKNP03, YWL$^+$20], a proof can be completed using only a small number of public key operations[1]; the remainder of the protocol requires only simple and highly efficient symmetric key operations. Finally, the GC-ZK paradigm places very low memory constraints on $\mathcal{P}$, which for many other protocols becomes a bottleneck (see e.g. discussion in [YSWW20]).

Our construction can be categorized as a GC-ZK technique. While we build on more recent arithmetic techniques (see next), the *ZEE* arithmetic technique that we build on is formalized in the GC-ZK framework proposed by [JKO13] and updated by [FNO15]. Moreover, the top-level protocol that hosts our implementation was formalized by [JKO13].

3.2 Arithmetic GC-ZK via IT-MACs

Earlier GC-ZK techniques, e.g. [JKO13, HK20b], worked directly with Boolean garbled circuits. These techniques were based on the classic GC technique of encoding Boolean functions as *encrypted truth tables*: given input labels, $\mathcal{P}$ decrypts the corresponding output label. To protect against a cheating $\mathcal{P}$, these techniques needed long labels: label length was proportional to the computational security parameter (e.g. 128 bits). Moreover, each label could only hold one semantic bit.

More recently, [HK20a] updated the GC-ZK technique by showing that it is possible to replace GC labels by simple information theoretic message authentication codes (IT-MACs). These IT-MACs are both shorter (e.g. 40 bits), since they are proportional only to the *statistical* security parameter, and also can hold a semantic arithmetic value with length equal to the length of the IT-MAC.

While our epoch parallelism technique is relatively agnostic to the low level details of the underlying protocol, ultimately we use the IT-MAC based *ZEE* protocol of [HYDK21]. Thus we briefly review the IT-MAC technique.

In the protocol, $\mathcal{P}$ and $\mathcal{V}$ hold IT-MACs that each encode a value in a field $\mathbb{Z}_p$ for a suitably large prime p (we choose $p = 2^{40} - 87$, the largest 40 bit prime). An IT-MAC consists of two *shares*, one held by $\mathcal{V}$ and one by $\mathcal{P}$. We denote the IT-MAC that encodes $x \in \mathbb{Z}_p$ by writing $[\![x]\!]$. This IT-MAC is a pair of values:

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

where $\mathcal{V}$ holds the left hand element and $\mathcal{P}$ holds the right hand element. Here, Δ is a global uniform non-zero value drawn by $\mathcal{V}$ at the start of the protocol and is unknown to $\mathcal{P}$.

Crucially, an IT-MAC is *unforgeable*: given $x\Delta - X$, $\mathcal{P}$ cannot reliably construct $y\Delta - X$ for $y \neq x$. She can do so only by guessing Δ, which succeeds with probability $\frac{1}{p-1}$.

At the same time, the parties can *operate* on IT-MACs. First, IT-MACs are *additively homomorphic*, as $[\![x]\!] + [\![y]\!] = [\![x + y]\!]$ (where the sum of two IT-MACs is defined to be the pointwise sum of their two parts). Second, [HK20a]

[1]The number of required *base* oblivious transfers, which require public-key cryptography, scale only with the security parameter.

showed that it is easy to multiply a vector of IT-MACs by a secret bit $b \in \{0, 1\}$ chosen by $\mathcal{P}$ via a single *oblivious transfer*. These two operations suffice to implement arbitrary arithmetic circuits. Thus these two operations, combined with the unforgeability property of the IT-MACs, mean that these primitives can implement arbitrary ZK proofs.

3.3 The *ZEE* Proof System

Based on IT-MAC ZK algebra (Section 3.2), [HYDK21] developed a full ZKP system, *ZEE*, that handles proofs expressed as ANSI C programs. Our core contribution is that we convert *ZEE* to an epoch parallel proof system that we call *EZEE* (Epoch parallel *ZEE*). We briefly explain the relevant parts of the *ZEE* approach.

[HYDK21] breaks the problem of proving statements written as C programs into two parts:

- The C program is compiled to a custom instruction set architecture (ISA) via a custom compiler. The ISA is relatively typical, with the notable inclusion of a distinguished *QED* instruction; if the program executes *QED*, then the proof accepts. The programmer writes ordinary C code to express their proof, while placing *QED* behind appropriate program conditions.

- The *ZEE* ISA is executed by a custom ZK implementation built on top of IT-MAC algebra. This implementation handles programs instruction by instruction, and includes many optimizations, such as an improved ZK RAM, called BubbleCache. Each program instruction is handled by an arithmetic circuit.

Following the notation of [HYDK21], we refer to the ISA as the *ZEE architecture* and to the implementation as the *ZEE microarchitecture*. Our approach uses the *ZEE* microarchitecture's instruction circuit directly. However, rather than running the entire program in sequence, we split the program into epochs and run a smaller sequence of instructions on each of a number of machines.

3.4 Notation and Security Model

- $\mathcal{P}$ is the prover. We refer to $\mathcal{P}$ by she/her.
- $\mathcal{V}$ is the verifier. We refer to $\mathcal{V}$ by he/him.
- $[n]$ denotes the sequence of integers $0...n-1$.
- ρ is the statistical security parameter, e.g. 40.
- κ is the computational security parameter, e.g. 128.
- $\mathbb{Z}_p$ is the field of integers modulo prime p.

We clarify our considered security model. Our protocol involves an arbitrarily large number of communicating machines, some controlled by $\mathcal{P}$ and some by $\mathcal{V}$. However, we view all of $\mathcal{P}$ machines as part of the prover $\mathcal{P}$ (symmetrically for $\mathcal{V}$). That is, although there are a large number of machines involved, there are still only two parties. Our protocol is secure against a malicious adversary that corrupts one of the two parties.

4 Technical Overview

In this section we present our approach with sufficient detail to understand our contribution.

To reiterate, the core idea of the *ZEE* proof system (Section 3.3, [HYDK21]) is to handle a proof expressed as a ISA program one instruction at a time. Each instruction is handled by an arithmetic circuit expressed using the IT-MAC-based ZK proof algebra of [HK20a]. *ZEE* then sequentially executes each program instruction and, at the end of the execution, outputs one if and only if the CPU is in a distinguished *QED* state.

Our approach leverages this same idea, except that we execute portions of the program in parallel across worker node machines. Our *EZEE* proof system (Epoch parallel *ZEE*) parallelizes the proof execution in three steps:

1. The *EZEE* prover $\mathcal{P}$ runs a non-cryptographic, cleartext version of the *ZEE* architecture using her witness. This cleartext version runs the *ZEE* instructions, but does not provide any cryptographic guarantees or communicate with the verifier $\mathcal{V}$. As $\mathcal{P}$ runs this cleartext execution, she periodically records *snapshots* of the simulated CPU state. These snapshots include the state of RAM, registers, and the program counter. Although this cleartext execution of the proof runs sequentially, it does not use cryptography and hence completes quickly.

2. Once all snapshots are recorded, $\mathcal{P}$ distributes the snapshots amongst a number of worker nodes. $\mathcal{V}$ similarly initializes corresponding worker nodes, and the two sets of nodes are grouped into pairs. Each pair of nodes then starts from the snapshot state and performs a ZK proof that guarantees to $\mathcal{V}$ the correct execution of all proof program instructions leading to the next snapshot. We refer to this partial proof as an *epoch*.

3. The parties then *glue* the epochs together. Specifically, $\mathcal{P}$ proves in ZK that for each epoch i, the ending CPU state matches the starting snapshot for epoch $i + 1$.

Once all epochs are completed and glued, the proof is finished. We show that this proof succeeds if and only if the original, sequential proof would have succeeded. Crucially, all subproofs in step (2) can be run in parallel, and hence the latency to finish the overall proof is greatly decreased.

Our presentation proceeds as follows:

- Observe that the above high level strategy is relatively agnostic of the details of the *ZEE* architecture: we simply need that it is possible to execute an instruction and to glue epochs together. Since these requirements are quite general, Section 5 begins by capturing the requirements formally through an abstraction we call a *Proof Interface Machine* (*PIM*). The simple yet key result is that, given a *PIM*, it is possible to *rewrite* the execution of a program into an equivalent epoch parallel version.

- With *PIM* defined, Section 6 presents a system design that describes how $\mathcal{P}$ and $\mathcal{V}$ can run a *PIM*-based ZK protocol across a cluster of machines. This design focuses on systems aspects of our parallelization of *ZEE*, and is not yet crypto-formal, since we do not yet specify the precise protocol run between the workers.

- In Section 7, we show that *ZEE* can be expressed as a *PIM* and then plug the resulting definition into our system design. We call the resulting proof system *EZEE* (Epoch parallel *ZEE*). We explain protocol-specific details that must be handled and give protocol-specific improvements to the glue step of epoch parallelism.

- Finally, Section 8 describes our C/C++ implementation and evaluates its performance when running C programs inside ZK.

5 Proof Interface Machine

ZKP protocols that handle arbitrary statements usually encode such statements as *circuits*; at the lowest level, our protocol (and the *ZEE* protocol we build on) is the same. To achieve a high degree of parallelism, our goal is to take as input a size $O(n)$ circuit and *transform* it into e new circuits of size $O(n/e)$. Each of these e circuits proves correctness of a portion (epoch) of the total proof execution. While we must additionally prove that the initial and final states of the e epochs are related, it is our intent that the epochs will be run in parallel. Thus, the total proof latency is greatly decreased.

In this section, we introduce the necessary formalisms to facilitate and formally discuss this circuit transformation. Since we focus on CPU emulation, we choose a CPU instruction as a unit of proof progress. We find it convenient to represent the *process* of proving as the execution of a state machine, whose states correspond to proof states, and whose transition function specifies how CPU instructions update the proof state.

Following [HYDK21], we separate the specification of the state machine (*architecture*, including ISA spec, plaintext state, etc.) from its implementation (*microarchitecture*, including ISA and RAM implementation, encoded state, etc.). Thus our state machine definition includes corresponding architectural and microarchitectural parts. The microarchitecural components, both functions and state, denoted by a bar, will be operated on by the underlying ZK protocol.

Definition 1 (*PIM*)**.** *A proof interface machine (PIM) consists of a space of architectural states State, a space of microarchitectural (encoded) states* $\overline{State}$*, a space of inputs* Σ*, and seven procedures (procedures annotated with a bar are microarchitectural):*

$$\begin{aligned}
\mathcal{T} &: (\Sigma \times \mathit{State}) \to \mathit{State} \\
\mathit{accept} &: \mathit{State} \to \{0,1\} \\
\mathit{extract} &: \overline{\mathit{State}} \to \mathit{State} \\
\overline{\mathcal{T}} &: ((n \in \mathbb{N}) \times \Sigma^n \times \overline{\mathit{State}}) \to \overline{\mathit{State}} \\
\overline{\mathit{accept}} &: \overline{\mathit{State}} \to \{0,1\} \\
\overline{\mathit{embed}} &: \mathit{State} \to \overline{\mathit{State}} \\
\overline{\mathit{match}} &: (\overline{\mathit{State}} \times \overline{\mathit{State}}) \to \{0,1\}
\end{aligned}$$

subject to the following three requirements:

$$\begin{aligned}
&\forall w \in \Sigma^n, \\
&\quad \mathit{extract} \circ \overline{\mathcal{T}}(n, w, \cdot) \circ \overline{\mathit{embed}} = \mathcal{T}^n(w, \cdot) \\
&\forall\ \overline{\sigma} \in \overline{\mathit{State}}, \\
&\quad \overline{\mathit{accept}}(\overline{\sigma}) \Leftrightarrow \mathit{accept}(\mathit{extract}(\overline{\sigma})) \\
&\forall\ \overline{\sigma}_0, \overline{\sigma}_1 \in \overline{\mathit{State}}, \\
&\quad \overline{\mathit{match}}(\overline{\sigma}_0, \overline{\sigma}_1) \Leftrightarrow \mathit{extract}(\overline{\sigma}_0) = \mathit{extract}(\overline{\sigma}_1)
\end{aligned}$$

where $\mathcal{T}^n(w, \cdot)$ *denotes the function that applies* $\mathcal{T}$ n *times by passing the ith character from* w *to the ith call to* $\mathcal{T}$*.*

We explain this definition informally. First, the functions $\mathcal{T}$ and *accept* specify the architecture: $\mathcal{T}$ specifies how the state transitions (e.g., $\mathcal{T}$ might handle a single processor cycle), while *accept* queries if the machine has reached an accepting state. In the context of ZK, providing inputs to the *PIM* such that *accept* outputs one means that the inputs together constitute a convincing witness.

The functions $\overline{\mathcal{T}}$ and $\overline{\mathit{accept}}$ specify the corresponding microarchitecture. Note that, while $\mathcal{T}$ specifies only a single step, $\overline{\mathcal{T}}$ *simultaneously* captures n steps of the state machine. This difference is needed because in ZK it is often useful for $\mathcal{P}$ to *look ahead* at her witness to improve efficiency. Thus, we group n steps together such that this lookahead is formally possible in the microarchitecture.

To enforce that the architecture and microarchitecture appropriately correspond, we introduce *extract* and $\overline{\mathit{embed}}$, which map between the two kinds of states. The *PIM* coherence conditions force correspondence: starting from corresponding input states, then taking n steps in both the architecture and microarchitecture results in corresponding output states. Moreover, the microarchitecture accepts if and only if the architecture accepts.

Note that we consider $\overline{\mathit{embed}}$ to be a microarchitectural function, because $\overline{\mathit{embed}}$ is needed to set up initial proof states inside the ZK protocol.

The procedure $\overline{\mathit{match}}$ is needed for epoch parallelism: we need a procedure that allows $\mathcal{V}$ to check that the output state of one epoch *matches* the input state to the subsequent epoch. Note that this cannot be achieved by a simple equality check, because the two states might be different; we only insist that the two microarchitectural states correspond (via *extract*) to the same architectural state.

Our epoch parallelism technique relies on the following simple observation. Let k, e be two natural numbers and let $n \triangleq k \cdot e$. Here, e denotes a number of *epochs* and k denotes the number of steps performed per epoch. Let $w_0, ..., w_{e-1}$ denote e length-k strings that together concatenate to w and let σ_0 be an initial state. Note the following trivial fact:

$$\begin{aligned} &\mathit{accept}(\mathcal{T}^n(w, \sigma_0)) \Leftrightarrow \\ &\left(\left(\bigwedge_{i \in [e]} \mathcal{T}^k\left(w_i, \sigma_i\right) = \sigma_{i+1}\right) \wedge \ \mathit{accept}(\sigma_e)\right) \end{aligned} \tag{1}$$

where each σ_i is the initial state of epoch i and σ_e is the final state. That is, to compute the final state $\mathcal{T}^n(\sigma_0)$, we can appropriately compute the $\mathcal{T}^k$ transition e times.

By Equation (1) and Definition 1, the following simple yet crucial lemma holds:

Lemma 1 (Epoch Parallelism)**.** *Let n, e, k be natural numbers such that $k \cdot e = n$. For all PIMs, the following fact holds: Let $\sigma_0 \in$ State be an initial PIM architectural state, let $w \in \Sigma^n$ be a witness, and let $w_{i \in [e]} \in \Sigma^k$ be e chunks of w (which together concatenate to w). Then:*

$$\begin{aligned} &\mathit{accept}(\mathcal{T}^n(w, \sigma_0)) \Leftrightarrow \\ &\left(\left(\bigwedge_{i \in [e]} \overline{\mathit{match}}\left(\overline{\mathcal{T}}\left(k, w_i, \overline{\sigma}_i\right), \overline{\sigma}_{i+1}\right)\right) \wedge \ \overline{\mathit{accept}}(\overline{\sigma}_e)\right) \end{aligned}$$

where for each $i < e$, $\sigma_{i+1} \triangleq \mathcal{T}^k(w_i, \sigma_i)$ and where for each $i < e$ $\overline{\sigma}_i \triangleq \overline{\mathit{embed}}(\sigma_i)$.

In other words, during the proof we need not compute the circuit $\overline{\mathcal{T}}(n, w, \sigma_0)$ but rather can compute e simpler subcircuits $\overline{\mathcal{T}}(k, w_i, \sigma_i)$. Each of these subcircuits does not depend on the output of any other, and so all e subcircuits can be executed in parallel. We break dependencies between these subcircuits by allowing the circuit to take as input intermediate machine states (via calls to $\overline{\mathit{embed}}$). Once the subcircuits have finished, we can complete the proof by demonstrating that the intermediate states between epochs are related by calls to $\overline{\mathit{match}}$: this call ensures that a cheating $\mathcal{P}$ cannot substitute some invalid state into the middle of the proof execution.

Note that *PIM does not* guarantee security properties of the resulting parallelized system; this must be proven separately. For simplicity of notation, in

Equation (1) and Lemma 1, we divide the computation into equal sized epochs; our formalisms trivially generalize to epochs of different sizes.

Applicability of *PIM*. *PIM* is *not* an attempt to build a general compiler that transforms an arbitrary ZK proof system into an epoch parallel ZK proof system. Instead, *PIM* formalizes a program transformation that introduces parallelism to proof statements; the underlying ZK protocol must leverage the introduced parallelism to improve performance. Proof systems that allow for parallel proofs of conjunctive statements can take advantage of the PIM. The *ZEE* proof system [HYDK21] implements *PIM* interface and can take advantage of conjunctive statements; we prove that the *EZEE*, an epoch-parallel version of *ZEE*, is a secure ZKP system. A general "parallelizing ZKP compiler" would require designing a crypto API, which underlying ZKP systems would need to satisfy. We believe this is a well-motivated significant separate undertaking.

6 System Design

In Section 5, we described a class of state machines that can be used to encode ZK proof statements and, crucially, we gave Lemma 1 which proves that such state machines can be parallelized. In this section, we use the *PIM* definition (Definition 1) to design a high level *system* on which parallel interactive ZK proofs can execute. Our design allows us to scale interactive ZK to clusters of machines.

Note that the given design is not yet crypto-formal, since we do not at this point give a particular ZK protocol. In Section 7 we plug the *ZEE* IT-MAC-based ZK protocol into our design, resulting in a secure ZKP system. While we do not prove a general statement for plugging in different interactive ZK protocols, we view the *PIM* and our system design as a template for design of parallel interactive ZK protocols.

In our design, both $\mathcal{P}$ and $\mathcal{V}$ instantiate one distinguished *main* node and a number of *worker* nodes. We refer to the $\mathcal{P}$ main node as $\mathcal{P}_{\text{main}}$ and to the $\mathcal{V}$ main node as $\mathcal{V}_{\text{main}}$. We refer to the ith $\mathcal{P}$ worker node (resp. $\mathcal{V}$ worker node) as $\mathcal{P}_i$ (resp. $\mathcal{V}_i$). Figure 1 depicts the high level interaction between these nodes. The roles of these components are as follows:

- $\mathcal{P}_{\text{main}}$ uses the *PIM* definition's transition function $\mathcal{T}$ to compute in cleartext the entire proof. As it runs, it periodically takes snapshots of the current state σ_i. Additionally, it records portions of its witness w_i. Note that, in general, taking a snapshot or recording the witness in a real system such as ours, is intricate; for example, recording the witness may involve capturing arbitrary interactions between a C program and the surrounding operating system. In general, recording the witness involves building a full *non-determinism log* (i.e. the extended witness) for the execution of the proof program. $\mathcal{P}_{\text{main}}$ sends σ_i and w_i to a worker node $\mathcal{P}_i$.
- Workers $\mathcal{P}_i$ and $\mathcal{V}_i$ together run an epoch of the program. Specifically, they consider a circuit that they together run inside the ZK protocol. The

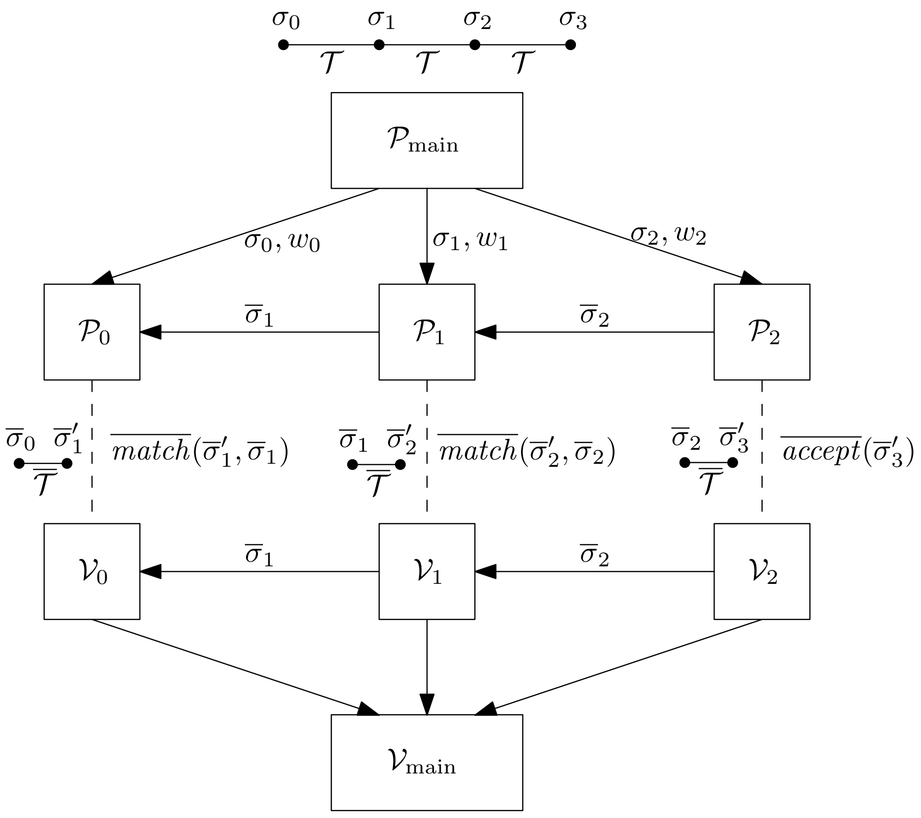

Figure 1: Our epoch parallel system design for three epochs. $\mathcal{P}_{\text{main}}$ first runs the proof locally by repeatedly calling $\mathcal{T}$; as she runs, $\mathcal{P}_{\text{main}}$ records snapshots of the proof state σ_i and of portions of her witness w_i. $\mathcal{P}_{\text{main}}$ sends σ_i and parts of her witness w_i to the prover workers $\mathcal{P}_i$. The prover worker $\mathcal{P}_i$ and verifier worker $\mathcal{V}_i$ then execute a ZK proof that demonstrates that (1) embedding σ_i to $\overline{\sigma_i}$ via a call to $\overline{embed}$, then running $\overline{\mathcal{T}}$ results in an embedded state $\overline{\sigma}'_{i+1}$ and (2) the embedded state $\overline{\sigma}_{i+1}$ (shares of which are sent back from the next workers) matches the ending state $\overline{\sigma}'_{i+1}$ via a call to $\overline{match}$. The final pair of workers instead call $\overline{accept}$ to check if the final proof state is accepting. Each $\mathcal{V}_i$ messages $\mathcal{V}_{\text{main}}$, indicating if its epoch succeeded or not. If all epochs succeed, $\mathcal{V}_{\text{main}}$ accepts the proof.

circuit performs the following actions: (1) take as input an intermediate state σ_i via a call to $\overline{embed}$, yielding encoded state $\overline{\sigma}_i$, (2) apply the transition function $\overline{\mathcal{T}}$ to map $\overline{\sigma}_i$ to a new state $\overline{\sigma}'_{i+1}$, (3) if the considered epoch is not the last one, check $\overline{match}(\overline{\sigma}'_{i+1}, \overline{\sigma}_{i+1}) = 1$ indicating that the ending state is equal to the starting state sent back from the next pair of workers, and (4) if the considered epoch *is* the last one, check $\overline{accept}(\overline{\sigma}'_{i+1}) = 1$ indicating that the final proof state is accepting.

- $\mathcal{V}_{\text{main}}$ waits to receive an accepting message from each worker $\mathcal{V}_i$. If each verifier worker accepts, $\mathcal{V}_{\text{main}}$ accepts the overall proof statement as true.

We note that when plugging in a specific ZK protocol, $\mathcal{P}_{\text{main}}$ and $\mathcal{V}_{\text{main}}$ can be leveraged to help coordinate the workers. This coordination can help deal with protocol-specific details. Looking forward, when we instantiate our system design in Section 7 with the protocol of [JKO13] (see Section 3.1), we use the main nodes to coordinate the required commitments and transmission of $\mathcal{V}$ randomness.

The execution of this system can be broken down into three stages: epoch generation, epoch parallel computation, and epoch verification. We next discuss these three stages in detail and explain the informal ZK protocol requirements needed to support each stage, with respect to both correctness and efficiency.

6.1 Epoch Generation

Recall that $\mathcal{P}_{\text{main}}$ first generates epochs by repeatedly calling $\mathcal{T}$. We refer to this process as *epoch generation*. To guarantee a correct and efficient system, epoch generation should meet several informal requirements:

- **Cheap To Generate** - Epoch generation occurs before any parallelization, running sequentially through the entire proof program. Thus, it is crucial that epoch generation completes quickly. I.e., running $\mathcal{T}$ in cleartext should be *much* (preferably orders of magnitude) faster than running $\overline{\mathcal{T}}$ inside the ZK protocol. Otherwise, proof latency will be constrained by the epoch generation step.

- **Deterministic** - Epochs must be deterministic. The system requires that, upon re-execution in the epoch parallel phase, each epoch reaches its predicted final state. This allows epochs to be glued together in the epoch verification phase. Therefore, the *PIM* microarchitecture must *perfectly* implement its corresponding architecture. Any deviation between the two will cause proofs to erroneously fail.

  To ensure epochs are deterministic, the system can record an epoch as a tuple of (1) a starting state, and (2) a non-determinism log. This requires the system to identify all non-determinism. I.e., the *PIM* must formalize all non-determinism as part of the witness w.

- **Equal Sized** - Epochs should be of roughly equal size. The entire proof cannot complete until each epoch finishes. If epochs are not of similar size, then any long-running epoch will become the bottleneck.

  For security, too, it is crucial that the size and other attributes of epochs (e.g., precisely the work performed) are independent of the witness, and hence of the proof execution flow. This is needed for simulation of the view of $\mathcal{V}$ in proving the ZK property. See Section 7.4.1 for discussion how using BubbleCache, the *ZEE* ORAM implementation, which allows cache misses, may be insecure in *EZEE*, and our resolution.

6.2 Epoch Parallel Computation

Once epochs have been generated, the system uses the ZKP protocol to ensure that each epoch is valid. This stage is the most computationally intensive portion of the system, as it executes the heavy work of actually performing the majority of the ZK computation: $\overline{\mathcal{T}}$. However, as the system has divided the computation into independent epochs, this work can be done in an embarrassingly parallel fashion. Namely, the system distributes epochs to arbitrarily large numbers of processors, even up to a cluster scale.

6.3 Epoch Verification

Once the system has proven that each epoch is valid, it then proves that, when combined, the epochs form a complete proof. This composition is achieved by calls to $\overline{\textit{match}}$. Note that even these calls to $\overline{\textit{match}}$ can be parallelized, since pairs of workers $\mathcal{P}_i$ and $\mathcal{V}_i$ calls $\overline{\textit{match}}$, not $\mathcal{P}_{\text{main}}$ and $\mathcal{V}_{\text{main}}$. After calling $\overline{\textit{embed}}$, a pair of workers immediately sends the encoded state back to their predecessors; thus $\overline{\textit{match}}$ can be called as soon as the call to $\overline{\mathcal{T}}$ is completed.

7 The *EZEE* Epoch Parallel Proof System

Section 5 introduced a formal framework for expressing epoch parallel ZK proofs, and Section 6 sketched a systems level design for running such ZK proofs across clusters of machines. However, as argued in Section 6, we could not directly prove the security of this design with respect to an arbitrary protocol. In this section, we formally instantiate our design with the *ZEE* protocol (see Section 3.3) and clarify interesting points that arise. We call the instantiated ZK protocol *EZEE*. Figure 2 illustrates many details of the *EZEE* protocol.

We begin by formally defining the *ZEE PIM* and the instantiated epoch parallel ZK protocol *EZEE*. The remainder of this section is dedicated explaining these constructions; Appendix A sketches a proof that *EZEE* is a secure ZKP system.

Recall from Section 5 that, to use our system, we must define seven procedures corresponding to the *PIM* definition. For *ZEE*, most of these definitions

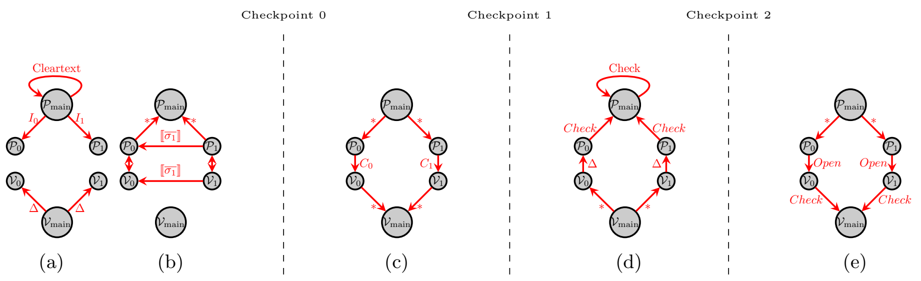

Figure 2: A two epoch *EZEE* execution. (a) $\mathcal{P}_{\text{main}}$ first executes the program in cleartext and collects the initial information I_0 and I_1 for both epochs. This information is sent to the prover workers $\mathcal{P}_i$. $\mathcal{V}_{\text{main}}$ generates the global IT-MAC secret Δ and distributes it to each verifier worker V_i. (b) Each pair of workers $\mathcal{P}_i$ and $\mathcal{V}_i$ begin a ZK proof starting from $\mathcal{P}_i$'s initial state I_i. $\mathcal{P}_1/\mathcal{V}_1$ transmit their encoded initial state $[\![\overline{\sigma_1}]\!]$ to $\mathcal{P}_0/\mathcal{V}_0$. Both pairs of workers execute their epoch in ZK until reaching the shared final states $[\![\overline{\sigma}'_1]\!]$ and $[\![\overline{\sigma}'_2]\!]$. While the epoch 1 workers prove that the *pc* terminated at *QED* (i.e. $\overline{accept}(\overline{\sigma}'_2) = 1$), the epoch 0 workers prove in ZK that $\overline{match}(\overline{\sigma}'_1, \overline{\sigma}_1) = 1$. After computing its output value, each $\mathcal{P}_i$ sends a signal to $\mathcal{P}_{\text{main}}$ and waits. (c) Once each $\mathcal{P}_i$ has indicated it has finished computing its part of the circuit (Checkpoint 0, Section 7.1), $\mathcal{P}_{\text{main}}$ instructs each worker $\mathcal{P}_i$ to commit to her proof output value. Each worker $\mathcal{P}_i$ sends a commitment C_i to $\mathcal{V}_i$. After receiving the commitment, each $\mathcal{V}_i$ sends a signal to $\mathcal{V}_{\text{main}}$ and waits. (c) After each $\mathcal{V}_i$ receives a commitment from $\mathcal{P}_i$ (Checkpoint 1), $\mathcal{V}_{\text{main}}$ instructs each $\mathcal{V}_i$ to share its randomness, including the global secret Δ_i, with $\mathcal{P}_i$. $\mathcal{P}_i$ then checks $\mathcal{V}_i$ did not cheat by replaying all of $\mathcal{V}_i$'s actions and checking that $\mathcal{V}_i$'s messages were well-formed. $\mathcal{P}_i$ forwards the result of this check and Δ_i to $\mathcal{P}_{\text{main}}$, then waits. $\mathcal{P}_{\text{main}}$ ensures that no V_i cheated. (Checkpoint 2) (c) Finally, $\mathcal{P}_{\text{main}}$ instructs each $\mathcal{P}_i$ to open its commitment. If each $\mathcal{V}_i$ successfully verifies the commitment, $\mathcal{V}$ is convinced that the overall proof is valid.

are inherited directly from [HYDK21], but $\overline{\textit{embed}}$ and $\overline{\textit{match}}$ remain to be defined. *ZEE* did not directly define these two procedures because they were not needed for a "single epoch" proof execution.

Construction 1 (*ZEE PIM*)**.** *The ZEE construction [HYDK21] implements the PIM interface (Definition 1) as follows:*

- *State is the space of ZEE architectural states. It includes a program counter, a program memory, a small registry, and a large main memory. Each memory is represented as a simple array of* 32*-bit values.*
- $\overline{\textit{State}}$ *is the space of ZEE microarchitecural states. It contains the same elements as the architectural states, but the representation is different. First, all values are elements in* $\mathbb{Z}_p$ *for prime* p *rather than* 32*-bit values. Second, each memory is represented by a construction called BubbleRAM*[2] *[HK20a] which maintains the memory in a permuted order (in practice, the permutation order is known to* $\mathcal{P}$ *and unknown to* $\mathcal{V}$*).*
- $\mathcal{T}$ *is defined by the ZEE ISA and handles the execution of a single ZEE instruction as defined in [HYDK21]. I.e.,* $\mathcal{T}$ *reads an instruction from memory and performs the corresponding state update, e.g. reading/writing to main memory, performing arithmetic, or jumping to a new program location. Note, if the architecture is in the distinguished QED state, then* $\mathcal{T}$ *is a no-op: the architecture waits in this state until accept is called.*
- *accept is defined with respect to ZEE's distinguished QED instruction: accept reads a final instruction from memory and checks if its op-code is QED. If so, accept outputs one; else it outputs zero.*
- $\overline{\mathcal{T}}$ *and* $\overline{\textit{accept}}$ *are the corresponding microarchitectural implementations of* $\mathcal{T}$ *and accept. This handling is defined in [HYDK21].*
- $\overline{\textit{embed}}$ *and* $\overline{\textit{match}}$ *are defined in Section 7.2.*
- *extract maps a ZEE microarchitectural state to a corresponding architectural state by reading entries from the microarchitectural memories and writing these values into a fresh architectural state. The key detail is that extract removes the permutation implied by BubbleRAM and writes the values into a simple array.*

The fact that these definitions satisfy the *PIM* coherence conditions follows from the completeness of the *ZEE* microarchitecture and from discussion about $\overline{\textit{embed}}$ and $\overline{\textit{match}}$ in Section 7.2.

We use Construction 1 to instantiate an epoch parallel ZK protocol:

[2]Technically, *ZEE*, as presented in [HYDK21], uses an improvement to BubbleRAM called BubbleCache. However, for our purposes we use BubbleRAM. We discuss this point at greater length in Section 7.4.1.

Construction 2 (*EZEE* Proof System)**.** *The EZEE proof system is the GC-ZK protocol [JKO13] instantiated with the ZEE PIM (Construction 1). The EZEE proof system takes as input a ZEE program. It applies Lemma 1 to transform the input program into an epoch parallel program. In the protocol, both $\mathcal{P}$ and $\mathcal{V}$ dispatch epochs to a set of workers. These workers pairwise execute their epoch as described in Section 7.1. Crucially, the workers execute their epochs* in parallel*. EZEE also dispatches more than one epoch to each worker (see Section 7.4.2); in this case, the workers execute their epochs sequentially. At certain steps of the protocol, $\mathcal{P}$ and $\mathcal{V}$ implement proof checkpoints (Section 7.1): i.e., they ensure that no worker proceeds to the next protocol step until* each *worker finishes the current step.*

We prove the following theorem, which proves that Construction 1 is a secure ZKP system, in Appendix A.

Theorem 1 (*EZEE* Security)**.** *Assuming a collision resistant hash function, that the prime modulus $p > 2^{37}$, and that $\lfloor \log p \rfloor \geq \rho$, Construction 2 is a* sound *(with soundness error $2^{-\rho}$) and* complete *malicious-verifier Zero Knowledge proof system that proves arbitrary ZK relations expressed as ZEE programs [HYDK21] in the OT-hybrid model.*

7.1 *EZEE* Protocol Checkpoints

Recall from Section 3 that the *ZEE* protocol of [HYDK21] is a GC-ZK-based protocol [JKO13]. We plug this protocol into the system design described in Section 6. We approach parallelism carefully, as subtle issues can emerge with parallel composition of standalone-secure protocols.

Our basic technique for ensuring security is to synchronize the workers' messages by introducing *checkpoints*: once a worker finishes a protocol step, it *waits* for its peers to catch up before proceeding. This ensures that we adhere to the *ZEE* message flow and the [JKO13] framework, as discussed next and in our proof of security. Serializing message flow allows for a trivial security reduction to [JKO13]. Indeed, the only step of the protocol that is run in parallel is the OT execution. We handle the security issues arising from parallelization by using a UC-secure [Can01] OT protocol (Ferret OT [YWL+20] was proved UC-secure by [WYKW21b]).

There are three *checkpoints* in *EZEE*.

***Checkpoint* 0:** OT checkpoint. In GC-ZK, after performing all OTs, the prover commits to her GC output label. Our first checkpoint preserves the ordering of messages in the GC-ZK protocol by ensuring that all (concurrently executed) OTs are completed before any worker commits to its output label.

Here and in other checkpoints $\mathcal{P}_{\text{main}}$ orchestrates the synchronization. In our presentation, workers $\mathcal{P}_i$ send the commitments (resp. other messages) directly to $\mathcal{V}_i$. One can think about them as being routed through $\mathcal{P}_{\text{main}}$ for even more explicit view of serialization.

***Checkpoint* 1:** commitment checkpoint. In GC-ZK, $\mathcal{V}$ sends to $\mathcal{P}$ all garbling randomness *after $\mathcal{P}$ commits her GC output label.* Similarly, in *EZEE*, each verifier worker send its randomness to its corresponding prover worker. Crucially, no verifier worker sends its randomness until *all* prover commitments are received. Without enforcing this, $\mathcal{P}$ learns $\mathcal{V}$'s global secret Δ for IT-MACs in advance, allowing $\mathcal{P}$ to forge proof values.

***Checkpoint* 2:** replay checkpoint. In GC-ZK, $\mathcal{P}$ must replay $\mathcal{V}$'s actions to ensure that all messages were properly constructed. Only once this check succeeds does $\mathcal{P}$ open her commitment. Similarly, in *EZEE*, a prover worker can only open its commitment once *every* prover worker finishes checking its epoch. As an additional detail, our $\mathcal{P}$ must make sure that each prover worker receives the same global IT-MAC secret Δ. This ensures that $\mathcal{V}$ cannot cheat during the $\overline{match}$ step of the proof.

7.2 *EZEE*'s $\overline{embed}$ and $\overline{match}$ Procedures

EZEE's $\overline{embed}$ procedure is straightforward: it takes as input an architectural state σ and constructs a microarchitectural state $\overline{\sigma}$ by choosing BubbleRAM permutations in any arbitrary manner; we later refine this choice of permutation.

$\overline{match}(\overline{\sigma}, \overline{\sigma}')$ checks that, if we account for the BubbleRAM permutations π and π' applied to the states, then the resulting values are equal. That is, $\overline{match}$ is defined as follows:

$$\overline{match}(\overline{\sigma}, \overline{\sigma}') \triangleq (\pi' \circ \pi^{-1})(\overline{\sigma}) \overset{?}{=} \overline{\sigma}'$$

where π is the BubbleRAM permutation of σ and π' is the BubbleRAM permutation of σ'.

In order to build our protocol, $\mathcal{P}$ and $\mathcal{V}$ workers must implement both $\overline{embed}$ and $\overline{match}$ as part of a secure ZK protocol. For example, $\overline{embed}$ uses OTs to allow $\mathcal{P}$ to select IT-MACs corresponding a particular input state under a particular permutation. Thus, the implementation of these procedures is potentially expensive. We introduce a simple trick that makes the implementation of $\overline{match}$ more efficient.

Our trick is based on the fact that for epoch parallelism (Lemma 1), we only call $\overline{match}$ in the case where one of the inputs is a freshly embedded input state. Thus, we adjust the definition of $\overline{embed}$ such that the permutation used to initialize BubbleRAM is chosen *uniformly.* This, in particular, ensures that in all cases where we call $\overline{match}$, the composed permutation $\pi' \circ \pi^{-1}$ is *also* uniform. Because of this, we need not compute the permutation $\pi' \circ \pi^{-1}$ *inside* the ZK circuit. Rather, $\mathcal{P}$ can securely reveal this uniform permutation to $\mathcal{V}$ without leaking any information. The two parties now *locally* permute their IT-MAC shares, achieving the permutation with essentially no cryptographic overhead. This saves significantly, since permuting inside a circuit requires a Waksman permutation network [Wak68], which, to permute n values, requires $O(n \log n)$ gates (and hence $O(n \log n)$ OTs in the *ZEE* protocol).

7.3 *EZEE*'s Main Nodes

As discussed in Section 6, *EZEE* must fully specify the four types of nodes. While prover worker and verifier worker essentially execute the *ZEE* PIM (with adjustments mentioned throughout this section), $\mathcal{P}_{\text{main}}$ and $\mathcal{V}_{\text{main}}$ must be specified.

7.3.1 *EZEE*'s $\mathcal{P}_{\text{main}}$

Recall from Section 6 that the epochs should be *cheap to generate*, *deterministic* and *equal sized*. $\mathcal{P}_{\text{main}}$ achieves these goals as follows.

Per our system design, $\mathcal{P}_{\text{main}}$ executes in cleartext the entire proof. This cleartext emulator does not model any cryptographic primitives used by *ZEE*, such as BubbleRAM. This fact is important, since the cleartext execution should finish as quickly as possible. A simple experiment shows that a more complex cleartext emulator that also models BubbleRAM is about $300\times$ slower than our faster emulator that does not model BubbleRAM. This emulator can be seen as a traditional CPU. As it runs, the emulator takes snapshots of intermediate states σ_i and calculates the number of needed program instructions.

Recall that $\mathcal{P}_{\text{main}}$ must also send to each worker $\mathcal{P}_i$ its partial witness w_i. In *ZEE*, $\mathcal{P}$'s entire witness is captured via calls to a distinguished `INPUT` instruction. As $\mathcal{P}_{\text{main}}$'s cleartext emulator runs, it captures the witness by recording a log of all `INPUT` instruction results. Note that the `INPUT` instruction is the *only* non-deterministic instruction in *ZEE*'s ISA. The partial witness w_i can be viewed as a non-determinism log, ensuring that each epoch is deterministic.

Once each epoch is generated, $\mathcal{P}_{\text{main}}$ passes the required information I_i to the corresponding prover worker $\mathcal{P}_i$. Besides σ_i and w_i, I_i also includes the number of instructions to be executed in this epoch, and includes two random seeds used to generate two uniform BubbleRAM permutations π_i and π_{i+1} as discussed in Section 7.2. Once $\mathcal{P}_i$ receives I_i, the worker pair can immediately begin executing its epoch.

$\mathcal{P}_{\text{main}}$ is also responsible for enforcing Checkpoints 0 and 2. This is simple; e.g, for Checkpoint 2, each prover worker $\mathcal{P}_i$ sends a bit indicating whether all messages from $\mathcal{V}_i$ were properly constructed. It also sends the global secret Δ_i provided by $\mathcal{V}_i$. $\mathcal{P}_{\text{main}}$ then checks that all received bits are one, and checks that all values Δ_i are equal. If so, it instructs every prover worker $\mathcal{P}_i$ to open its commitment to $\mathcal{V}_i$.

7.3.2 *EZEE*'s $\mathcal{V}_{\text{main}}$

EZEE's $\mathcal{V}_{\text{main}}$ is responsible for ensuring that each verifier worker $\mathcal{V}_i$ is convinced of the validity of epoch i such that all subproofs can be stitched into a complete proof. $\mathcal{V}_{\text{main}}$ also distributes the global secret Δ to each verifier worker $\mathcal{V}_i$.

$\mathcal{V}_{\text{main}}$ is responsible for enforcing Checkpoint 1. That is, each verifier worker $\mathcal{V}_i$ sends a signal to $\mathcal{V}_{\text{main}}$ once it receives a commitment from $\mathcal{P}_i$. Only once all such signals are received, does $\mathcal{V}_{\text{main}}$ instruct each $\mathcal{V}_i$ to send its randomness to

$\mathcal{P}_i$. This ensures that all commitments are received before revealing any of $\mathcal{V}$'s secret randomness.

7.4 Additional Modifications

7.4.1 Potential Leakage due to BubbleCache

By default, *ZEE* uses BubbleCache [HYDK21] as its ZK RAM. BubbleCache improves RAM performance by allowing for *cache misses*. Whenever a cache miss occurs, *ZEE* simply executes no-op instructions, allowing BubbleCache to catch up. Thus, in *ZEE* equipped with BubbleCache, it is likely that the number of instructions will differ from the number of needed processor cycles.

In *EZEE*, $\mathcal{P}_{\text{main}}$ runs its cleartext emulator without modelling the ZK RAM (see Section 7.3.1), and epochs generated by $\mathcal{P}_{\text{main}}$ contain equal numbers of instructions. However, because of the cache miss feature of BubbleCache, two epochs with same number of instructions may require different numbers of cycles. In other words, if *EZEE*'s workers use *ZEE* with BubbleCache and each worker executes the same number of instructions, $\mathcal{V}$ is able to learn the cache miss rate distribution across epochs, which is not possible in *ZEE* without epoch parallelism, cannot be simulated and is not secure.

Therefore, we replace BubbleCache by its predecessor BubbleRAM. BubbleRAM does not allow cache misses, so the number of instructions matches precisely the number of needed processor cycles. Even in extreme scenarios BubbleRAM, as compared to BubbleCache, reduces *ZEE* cycle performance by at most around 30% [HYDK21]. Moreover, any such overhead is fully parallelized in our system.

7.4.2 Reducing Memory Consumption

The *ZEE* implementation consumes physical memory proportional to the number of executed instructions. Thus, for very long running proofs, physical memory becomes a serious concern. Our experiments show that *ZEE*'s $\mathcal{P}$ and $\mathcal{V}$ each require over 22GB of physical memory when executing 1 million instructions using a 2^{22} word ZK RAM. Therefore, naïvely dividing a large execution into epochs might still exhaust the available hardware resource.

Fortunately, our epoch parallelism technique solves this problem without needing to significantly re-engineer *ZEE*. Our idea is to allocate more than one epoch to a single hardware device. This device executes each of its epochs in sequence. More specifically, given d devices, we execute on the ith device epochs i, $i + d$, $i + 2d$, etc. Since each epoch runs only a portion of the program, the workers only need enough memory for that portion. By increasing the number of epochs, we decrease the size of program portions and reduce per-epoch memory consumption.

This does not yet completely resolve the issue, since no epoch can proceed past Checkpoint 1 until *all* epochs reach Checkpoint 1 (Section 7.1). However, we ensure that the per-epoch amount of memory that must be stored across

Checkpoint 1 is small. Thus, each device runs each of its epochs up to Checkpoint 1; when the checkpoint is reached, the device simply reuses its physical memory to handle its next epoch.

Achieving constant storage across Checkpoint 1 is non-trivial. Specifically, after Checkpoint 1, $\mathcal{P}_i$ must check that all messages received from $\mathcal{V}_i$ were properly constructed (Section 3.1). Thus, naïvely, $\mathcal{P}_i$ must store all messages received from $\mathcal{V}_i$, which are together very large. This can be easily resolved by computing a hash digest of all messages received from $\mathcal{V}_i$. Upon receiving $\mathcal{V}_i$'s randomness, $\mathcal{P}_i$ reconstructs the messages, computes a new hash digest, and checks it is equal to the stored constant size digest.

A more difficult problem is in ensuring that $\mathcal{V}_i$'s randomness can be compactly represented. *ZEE* uses the recent Ferret correlated OT protocol [YWL+20]. For each of her input bits b, correlated OT ensures that $\mathcal{P}$ receives either a uniform value $X \in \{0,1\}^\kappa$ or $X \oplus R$ where R is a fixed global value. Crucially, each value X is chosen by the OT protocol. Hence, $\mathcal{P}$ cannot locally expand each value X starting from a compact PRG seed without replaying the OT protocol in her head. We do not do this because the bottleneck in Ferret performance is computation. We instead alter the OT protocol such that each value X can be simply derived from a PRG seed. Specifically, when Ferret OT sends to $\mathcal{P}$ $Y \oplus bR$, $\mathcal{V}$ also sends $X \oplus Y$ such that $\mathcal{P}$ can compute $X \oplus bR$. Thus, $\mathcal{P}$ can reconstruct all OT values, and hence all messages, by expanding a constant sized PRG seed. The technique does require added communication, but is needed to allow for long running proofs.

By the above adjustments, each worker needs to store only a small constant amount of information (specifically, digests and PRG seeds) across Checkpoint 1.

8 Evaluation

In this section, we describe our *EZEE* implementation and then we evaluate its performance. Our evaluation focuses on *EZEE*'s *proof latency*, i.e. the total end-to-end proof runtime. We compare *EZEE* to the non-parallel *ZEE* proof system, and we give cost breakdowns of the different *EZEE* components.

8.1 Implementation

We implemented *EZEE* in C/C++ based on *ZEE*. Prover and verifier workers are implemented on top of *ZEE*'s backend cryptographic ZK protocol. We added around 800LOC to account for *EZEE* specific concerns, such as gluing and checkpointing. $\mathcal{P}_{\text{main}}$ and $\mathcal{V}_{\text{main}}$ are implemented in around 1400LOC. We used *ZEE*'s frontend compiler and standard library to compile our benchmarks.

Data Availability. We plan to open-source this project to the community.

Benchmark	**# *ZEE* Instrs.**	**Mem. Words**	***ZEE* Latency**
sed bug	344,051	2^{13}	31.8s
bzip2, 8.1KB image	169,353,341	2^{22}	4h 20m*
bzip2, 278.5KB image	5,000,578,992	2^{22}	5d 8h 15m*
bzip2, 652.3KB image	11,859,715,862	2^{22}	12d 16h 12m*

Figure 3: Benchmark summary. Proof latency for the *sed* bug experiment were obtained by running *ZEE* without epoch parallelism; proof latency for the *bzip2* experiments (i.e. those marked with *) were estimated based on *sed*'s execution speed because *ZEE* cannot run these long experiments.

Benchmark	***ZEE* Latency**	**Adjusted Baseline Latency**	***EZEE* Latency**	**Speedup**	***EZEE* CPU Clock Rate**
sed bug	31.8s	N/A	2.8s	11.4×	122.9KHz
bzip2, 8.1KB image	4h 20m*	3h 33m 28s	8m 20s	25.6×	338.4KHz
bzip2, 278.5KB image	5d 8h 15m*	4d 9h 58m 52s	3h 31m 15s	30.1×	394.5KHz
bzip2, 652.3KB image	12d 16h 12m*	10d 4h 46m 16s	8h 37m 9s	28.4×	382.2KHz

Figure 4: Experimental results. *ZEE* proof latency is explained in Figure 3. Measurements for *ZEE* (i.e. those marked with *) were estimated based on *sed* performance. The adjusted baseline system is explained in Section 8.3. We list *EZEE*'s total proof latency, its speedup over the adjusted baseline (except for *sed*, which is instead compared to *ZEE* directly), and its clock rate.

8.2 Environment and Benchmarks

We evaluated *EZEE* on Cloudlab [DRM+19]. We used a network of c6525-25g machines: 16-core AMD 7302P at 3.00GHz, 128GB ECC Memory, two dual-port Mellanox ConnectX-5 25Gbps NIC, connected via a central Dell Z9332 switch and multiple Dell S5296F switches to form a star network. See [CLO] for precise server and network specification.

Due to large number of Cloudlab users, we were only able to allocate 62 of these nodes. These nodes were arranged as follows: 1 for $\mathcal{P}_{\text{main}}$, 1 for $\mathcal{V}_{\text{main}}$, 30 for prover workers $\mathcal{P}_i$, and 30 for verifier workers $\mathcal{V}_i$. Since one *EZEE* worker requires significant physical memory, we used only two cores per machine. Thus we can allocate a maximum of 60 worker cores such that each core has 64GB of physical memory.

We evaluated our system with two Linux programs:

bzip2 is a benchmark in SPEC2006 [Hen06], an industry-standard, CPU-intensive benchmark suite. We used *bzip2* to compress three different-sized images. Two are taken from the SPEC2006 input data set. We prove in ZK that the program terminates normally. This benchmark, in part, demonstrates that *EZEE* can achieve long running proofs that were previously impossible with the unmodified *ZEE* system. For this benchmark, we instantiate a 2^{22} word ZK RAM.

sed 1.17 contains a segmentation fault bug listed in the Software-artifact Infrastructure Repository (SIR) [DER05]. Specific inputs cause this version of *sed* to invoke the standard function *memmove* to attempt to move -1 bytes of memory, leading to a segmentation fault. [HYDK21] showed that *ZEE* can prove the existence of this bug in ZK. We used *EZEE* to achieve the same proof with lower proof latency. As per [HYDK21], we run this benchmark with a 2^{13} word ZK RAM.

Figure 3 summarizes information about our benchmarks.

8.3 Baseline Evaluation

Ideally, we would use off-the-shelf *ZEE* as a point of comparison for our benchmarks. Unfortunately, this is not possible because *ZEE* consumes physical memory proportional to the proof runtime. Thus, when we tried to use *ZEE* to execute our long running *bzip2* benchmarks, we exhausted all available physical memory and were unable to complete the proof. We *were* able to fully execute the much shorter *sed* benchmark. Figure 3 tabulates *ZEE*'s *sed* performance and uses this value to extrapolate *bzip2* performance.

However, we still wish to have a point of comparison that is not based on extrapolation. In Section 7.4.2, we explained that epochs can be used to help reduce memory consumption. We build a non-parallel baseline system by instantiating *EZEE* using only one worker pair. These two workers sequentially execute each epoch and thus emulate an unmodified *ZEE* system that can handle much longer proofs. To more closely capture the performance of the unmodified *ZEE* system, we instantiate this baseline using BubbleCache rather than BubbleRAM. Technically, this is not secure (see discussion in Section 7.4.1), but *ZEE* can securely use BubbleCache, and we only want a performance estimate for *ZEE*. Note, our parallel *EZEE* implementation uses BubbleRAM and is secure.

We tabulate the performance of this *adjusted baseline* system in Figure 4. Note that the adjusted baseline system incurs the glue overhead of *EZEE* (i.e. calls to $\overline{match}$, $\overline{embed}$, and a cleartext emulator). Nevertheless and surprisingly, the adjusted baseline outperforms the extrapolated performance of *ZEE*. Performance is improved because *ZEE* performance was extrapolated from the very short *sed* proof. Each epoch run by our adjusted baseline is significantly longer than the entire *sed* proof (around six million instructions per epoch). As proofs run longer, the *ZEE* architecture amortizes some costs, e.g. accesses to BubbleCache.

8.4 End-to-end Proof Latency

We first evaluate our end-to-end proof latency when using the available 60 cores. End-to-end latency is measured starting from $\mathcal{P}_{\text{main}}$'s initialization and ending when $\mathcal{V}_{\text{main}}$ accepts the proof. Figure 4 tabulates these experimental results. As compared to our adjusted baseline, *EZEE* accelerates *bzip2* by approximately 28×. The resulting CPU runs at up to 394KHz. Our improvement to *sed* is less

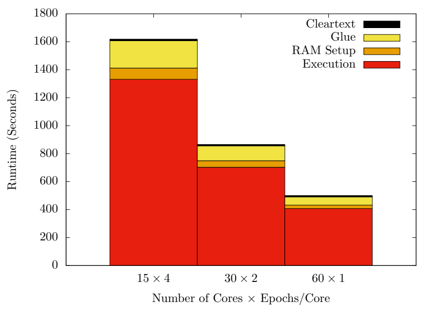

Figure 5: *EZEE* proof latency decomposition for the *bzip2* benchmark when compressing an 8.1KB image. Note that because this proof requires large numbers of instructions, as we decrease the number of cores, each core is responsible for more epochs (see Section 7.4.2).

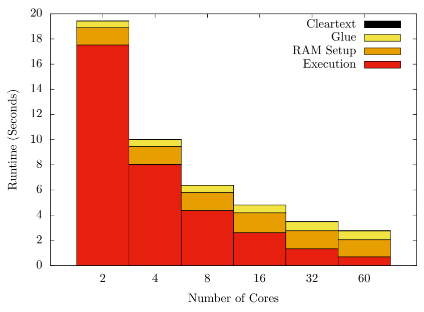

Figure 6: *EZEE* proof latency decomposition for the *sed* benchmark. Each core is responsible for only one epoch.

impressive, since this proof is very short. Hence, the non-parallelizable costs of our system begin to dominate and reduce our improvement. Nevertheless, we still accelerate *sed* by about 11.4×.

We next evaluate how *EZEE* performance scales with the number of cores. We ran the *bzip2* benchmark to compress an 8.1KB image while varying the number of utilized cores. Figure 7 plots performance. Recall that, as discussed in Section 7.4.2, we use epochs to reduce memory consumption. For this *bzip2* benchmark, we require 60 epochs to complete the proof without exhausting memory. Thus, as we reduce the number of cores, we must assign more epochs to each core. Some proof overhead increases proportionally with number of epochs per core (see next).

8.5 Proof Latency Breakdown

Next, we break down the costs of our system to identify the proof latency bottlenecks and the maximum possible performance. Although *EZEE* parallelizes the most expensive proof steps, there are still sequential components that cannot be avoided. We decompose proof latency into three major parts:

- **Execution:** Each pair of cores must finish its proof execution, i.e., $\overline{\mathcal{T}}$. The incurred latency is proportional to the number of instructions that each core pair executes.
- **Glue (and RAM setup):** Every worker pair must run $\overline{embed}$ and $\overline{match}$ in ZK. Crucially, the call to $\overline{embed}$ involves initializing BubbleRAM with a uniformly permuted RAM state (see Section 7.2), the most expensive

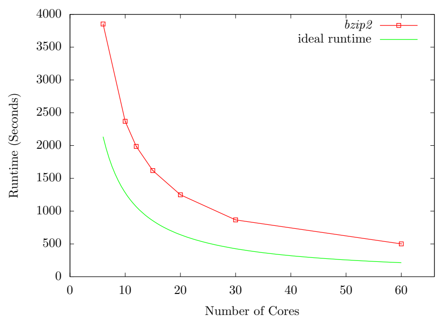

Figure 7: We used *bzip2* to compress an 8.1KB image inside ZK and measured proof latency as a function of the number of cores. For n cores, each core runs $\frac{60}{n}$ epochs. Ideal runtime is derived from the adjusted baseline runtime (Section 8.3).

portion of the glue step. The incurred latency is proportional to the number of epochs that each core executes.

- **Cleartext:** *EZEE*'s $\mathcal{P}_{\text{main}}$ must finish epoch generation before the parallel parts of the protocol begin. The incurred latency scales with the total proof runtime and with the number of epochs.

Figures 5 and 6 depict breakdowns of the above latency costs for different numbers of cores and for (1) *bzip2* with an 8.1KB image and (2) *sed*. Our plots separate the cost to initialize BubbleRAM from other glue step costs.

These plots show how *EZEE*'s latency decreases as more cores are added. The expensive execution step is made fast with large numbers of cores. Indeed for *sed* with 60 cores, glue, not execution, dominates in terms of latency. Given more cores, we could further accelerate *EZEE* for the *bzip2* benchmark: based on the cost of glue, we calculate that the maximum possible clock rate for this benchmark is ≈ 1.8MHz.

8.6 Network Traffic

Our cores communicate with one another through TCP/IP channels as implemented by [WMK16]. Specifically, each prover worker $\mathcal{P}_i$ maintains channels with $\mathcal{P}_{\text{main}}$, $\mathcal{P}_{i+1}$ and $\mathcal{V}_i$ (and symmetrically for verifier worker $\mathcal{V}_i$). We measured network traffic on these channels. Figure 8 tabulates the results. Unsurprisingly, traffic is light except between pairs of worker nodes. These workers communicate via large numbers of oblivious transfers and hence consume significant bandwidth.

Benchmark	$\mathcal{P}_{\text{main}}$ and $\mathcal{P}_i$	$\mathcal{P}_i$ and $\mathcal{P}_{i+1}$	$\mathcal{P}_i$ and $\mathcal{V}_i$	$\mathcal{V}_i$ and $\mathcal{V}_{i+1}$	$\mathcal{V}_{\text{main}}$ and $\mathcal{V}_i$
sed	33.5KB	129KB	181MB	129KB	608B
bzip2	16.2MB	64.1MB	37.2GB	64.1MB	608B

Figure 8: Bandwidth consumption of different pairs of nodes.

Acknowledgment

This work was supported in part by NSF award #1909769, by a Cisco research award, by Georgia Tech's IISP cybersecurity seed funding (CSF) award. This material is also based upon work supported in part by DARPA under Contract No. HR001120C0087. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References

[AAC+17] Hamza Abusalah, Joël Alwen, Bram Cohen, Danylo Khilko, Krzysztof Pietrzak, and Leonid Reyzin. Beyond hellman's time-memory trade-offs with applications to proofs of space. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part II*, volume 10625 of *LNCS*, pages 357–379. Springer, Heidelberg, December 2017.

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

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

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

[BCTV14a] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In

Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 276–294. Springer, Heidelberg, August 2014.

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

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

[BMRS20] Carsten Baum, Alex J. Malozemoff, Marc Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for arithmetic circuits with nested disjunctions. Cryptology ePrint Archive, Report 2020/1410, 2020. `https://eprint.iacr.org/2020/1410`.

[Can01] R. Canetti. Universally composable security: a new paradigm for cryptographic protocols. *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, 2001.

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

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

[CLO] CloudLab Documentation. `http://docs.cloudlab.us/hardware.html`. Retrieved Sept. 20, 2021.

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

[DIO20] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-point zero knowledge and its applications. Cryptology ePrint Archive, Report 2020/1446, 2020. `https://eprint.iacr.org/2020/1446`.

[DRM$^+$19] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[EFKP20] Naomi Ephraim, Cody Freitag, Ilan Komargodski, and Rafael Pass. Sparks: succinct parallelizable arguments of knowledge. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 707–737. Springer, 2020.

[FKL$^+$21] Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang, and Chenkai Weng. Constant-overhead zero-knowledge for ram programs. Cryptology ePrint Archive, Report 2021/979, 2021. `https://ia.cr/2021/979`.

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

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

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In Richard E. Ladner and Cynthia Dwork, editors, *40th ACM STOC*, pages 113–122. ACM Press, May 2008.

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

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

[Hen06] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

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

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

[HYDK21] David Heath, Yibin Yang, David Devecsery, and Vladimir Kolesnikov. Zero knowledge for everything and everyone: Fast ZK processor with cached ORAM for ANSI C programs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1538–1556. IEEE, 2021.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, August 2003.

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

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

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

[KP17] Yashvanth Kondi and Arpita Patra. Privacy-free garbled circuits for formulas: Size zero and information-theoretic. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 188–222. Springer, Heidelberg, August 2017.

[NVCF08] Edmund B Nightingale, Kaushik Veeraraghavan, Peter M Chen, and Jason Flinn. Rethink the sync. *ACM Transactions on Computer Systems (TOCS)*, 26(3):1–26, 2008.

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

[QDCF16] Andrew Quinn, David Devecsery, Peter M Chen, and Jason Flinn. Jetstream: Cluster-scale parallelization of information flow queries. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 451–466, 2016.

[SKW$^+$10] Martin Süßkraut, Thomas Knauth, Stefan Weigert, Ute Schiffel, Martin Meinhold, and Christof Fetzer. Prospect: A compiler framework for speculative parallelization. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code generation and Optimization*, pages 131–140, 2010.

[VLW$^+$12] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M Chen, Jason Flinn, and Satish Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. *ACM Transactions on Computer Systems (TOCS)*, 30(1):1–24, 2012.

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

[WDC$^+$13] Benjamin Wester, David Devecsery, Peter M Chen, Jason Flinn, and Satish Narayanasamy. Parallelizing data race detection. *ACM SIGARCH computer architecture news*, 41(1):27–38, 2013.

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

[WYKW21a] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *42nd IEEE Symposium on Security and Privacy*, 2021.

[WYKW21b] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[WZC$^+$18] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. DIZK: A distributed zero knowledge proof system. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 675–692, 2018.

[YSWW20] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. Cryptology ePrint Archive, Report 2021/076, 2020.

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

[ZS02] Craig Zilles and Gurindar Sohi. Master/slave speculative parallelization. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002.(MICRO-35). Proceedings.*, pages 85–96. IEEE, 2002.

A *EZEE* Proof Sketch

We sketch a proof of Theorem 1:

Proof Sketch. Note, the assumption of a collision resistant hash function and the requirement that $p > 2^{37}$ are inherited from *ZEE* security proof.

At a high level, our system simply implements a GC-ZK proof system, as proved secure by [JKO13]. However, our system does introduce concurrent OTs which must be properly handled.

We argue six main points:

First, we argue that we can apply our *PIM* circuit transformation to the *ZEE* proof system. This fact follows directly from Lemma 1.

Second, note that our explicit checkpoints (Section 7.1) precisely preserve the message ordering of the original *ZEE* protocol, except that the OTs are executed concurrently by $\mathcal{P}_i - \mathcal{V}_i$ pairs. Therefore, for now ignoring the parallel execution of OTs, our protocol is clearly secure under the [JKO13] framework: we simply run the *ZEE* protocol on a different – but equivalent – circuit.

Third, we show that it is safe to execute OTs concurrently in our protocol. This follows from two points:

1. All OT inputs from both $\mathcal{P}$ and $\mathcal{V}$ are defined before the first OT is issued.
2. The chosen Ferret OT protocol [YWL$^+$20,AAC$^+$17] is UC-secure [Can01].

Fourth, notice that $\mathcal{P}$ can now see the output of some OTs (i.e. those OT outputs corresponding to one epoch) before choosing her input for other OTs (i.e. those OT inputs corresponding to another epoch). This does not help a corrupt $\mathcal{P}$ *in our protocol*, since the received labels are all uniformly random and independent from each other. (Recall that in the [JKO13] protocol, $\mathcal{V}$ does open all such randomness, but this step is not done until all OTs are finished – cf Checkpoints 0 and 1). We stress that for general protocols, e.g., where $\mathcal{V}$'s OT inputs may be related to each other, this may not be secure.

Fifth, note that interleaved with these OTs, $\mathcal{P}$ sends to $\mathcal{V}$ uniform permutations per our implementation of $\overline{\textit{match}}$ (Section 7.2). That is, before performing the OTs for a given epoch, $\mathcal{P}_i$ sends to $\mathcal{V}_i$ the permutation $\pi_{i+1} \circ \pi_i^{-1}$. Similarly to the above point, this extra message preserves ZK because the starting permutation for epoch $i+1$ π_{i+1} is chosen uniformly, so the composed permutation is also uniform and conveys no useful information to a corrupt $\mathcal{V}$; it is easily simulatable by a ZK simulator. Alternatively, we can view these uniform permutations as part of the proved statement, established before the execution of the protocol, and hence treated as public knowledge.

Sixth, we note that our breakdown of the computation into epochs is independent of $\mathcal{P}$'s witness, and hence can be easily simulated given the program runtime. We recall that, as discussed in Section 7.4.1, this prevents us from using BubbleCache [HYDK21].

Therefore, our *EZEE* protocol is secure by reduction to the [JKO13] proof system.

EZEE is a secure ZKP protocol.

$\square$