CoBBl: Dynamic constraint generation for SNARKs

Kunming Jiang, Fraser Brown, and Riad S. Wahby
Carnegie Mellon University

General-purpose probabilistic proof systems operate on programs expressed as systems of arithmetic constraints—an unfriendly representation. There are two broad approaches in the literature to turning friendlier, high-level programs into constraints suitable for proof systems: direct translation and CPU emulation. Direct translators compile a program into highly optimized constraints; unfortunately, this process requires expressing all possible paths through the program, which results in compile times that scale with the program's runtime rather than its size. In addition, the prover must pay the cost of every possible program path, even those untaken by a given input. In contrast, CPU emulators don't compile programs to constraints; instead, they "execute" those programs, expressed as CPU instructions, on a CPU emulator that is itself expressed as constraints. As a result, this approach can't perform powerful, program-specific optimizations, and may require thousands of constraints where direct translation could use a clever handful. Worse, CPU emulators inherit an impractically expensive program state representation from the CPUs they emulate.

This paper presents a compiler and proof system, CoBBl, that combines the benefits of CPU emulation and direct translation: it takes advantage of program-specific optimizations, but doesn't pay for an unnecessary state representation or unexecuted computation. CoBBl outperforms CirC, a state-of-the-art direct translator, by 1–30× on compile time and 26–350× on prover time, and outperforms Jolt, a state-of-the-art CPU emulator, on prover time by 1.1–1.8× on Jolt-friendly benchmarks, and up to 100× on other benchmarks.

1. Introduction

People are *really excited* about SNARKs [14, 17, 37, 41, 44]. A SNARK is cryptographic protocol that allows a fast but untrusted prover, $\mathcal{P}$, to convince a slow but trusted verified $\mathcal{V}$ that it has correctly executed program on a certain set of inputs. This class of protocols motivated an entire DARPA program [73]; it's one reason Andreesen Horowitz, a prominent venture capital firm, hired a research team [1, 74]; and it's the backbone of companies like Starkware, which raised 100 million dollars at an 8 billion dollar valuation in 2022 [16], and Matter Labs, which has raised a total of $458 million over several years [79].

Despite the money that's poured into SNARK research from both academia and industry, building applications on top of SNARKs still requires both a special application and a special builder. To encode a program so that a proof system can operate on it, an expert developer must express that program in finicky, low-level arithmetic constraints; naively encoding an arbitrary program will almost certainly lead to unbearably slow prover times. Experts spend thousands of hours hand-tuning the constraints that encode even particularly well-suited applications. Matter Labs's ZKSync [4], for example, is designed to be proof friendly, was built by a team of professionals, and required several audits—including one that discovered a potential billion-dollar bug [57].

There are two general approaches to making SNARKs programmable for non-experts: CPU emulation and direct translation. Direct translators (e.g., [50, 62, 67, 70]) are custom compilers that consume high-level program descriptions and turn them into cleverly optimized constraints. In contrast, systems using the CPU emulator approach (e.g., [9, 13, 22, 75]) encode a CPU emulator in constraints. Proving that a program has executed correctly just requires supplying the emulator with the program—usually produced by an off-the-shelf compiler—and its inputs. Unfortunately, both the CPU emulator approach and the direct translator approach have downsides—downsides severe enough to prevent SNARKs from going mainstream, despite serious financial and intellectual investment. CPU emulation unnecessarily uses a CPU's (expensive) representation of program state, while direct translation unnecessarily executes (expensive) irrelevant computation; we explore these downsides further in the next paragraphs.

The difference between the two approaches, CPU emulation and direct translation, boils down to the difference between their *execution models*. Conceptually, an execution model is the interface between a program's semantics and its execution substrate. When compilers target a given processor, for example, they produce a representation that uses the processor's execution model (e.g., its instruction set and calling conventions) to express the program's semantics.

Direct translators produce a program representation whose execution model is exactly the circuit-like constraint formalism exposed by the proof system. This means program constructs that can be efficiently expressed in constraints are cheap. It also means that the compiler can use clever optimizations that take advantage of the constraint formalism's inherent nondeterminism [62] (§2.1).

The constraint formalism also dictates that constraints must be fixed at compile time, and thus must express *every possible execution path through the program*—including both branches of every conditional statement, every loop iteration up to a static bound, etc. As a result, unlike a

traditional compiler, the direct translator's compile time is proportional to the *runtime* of the program, not the size of the program description. Moreover, since the resulting constraints express every possible program execution, the prover must pay this cost—regardless of which path the program inputs actually take.

CPU emulators, on the other hand, provide exactly the execution model exposed by the emulated CPU (which is itself implemented in the constraint formalism). This means that CPU emulators do not pay for untaken program paths: the emulator is an unrolled fetch-decode-execute loop, so from the constraints' perspective the computation is always just a straight-line sequence of CPU state transitions. Moreover, CPU emulators can (and in practice do) use off-the-shelf compilers, meaning they can easily reuse existing tooling and support a range of programming languages.

A critical downside of CPU emulation, however, is that the CPU's execution model represents program state using a combination of registers, the call stack, and heap memory. This program state representation—especially the call stack, which entails vast numbers of loads and stores—results in massive overheads when implemented on top of a proof system. Furthermore, the emulator's execution model dictates that state transitions are only possible through a fixed set of processor instructions—which leaves clever constraint optimizations on the table.

Our key insight is that, through careful co-design of a compiler and proof machinery, it's possible to create an execution model *specifically optimized for proof systems*. The resulting system, CoBBL, captures the benefits (and avoids the drawbacks) of both direct translation and CPU emulation. Like a CPU emulator, CoBBL doesn't force the prover to pay for untaken branches and unnecessary loop iterations—and its compile time scales with the size of the program's representation, not its runtime. Meanwhile, like a direct translator, CoBBL avoids the overhead of CPU-like state management, and takes advantage of powerful, program-specific optimizations.

The CoBBL compiler breaks input programs into basic blocks, optimizes those blocks, and emits optimized constraints for each block. Then, the CoBBL prover proves that it has correctly executed a specific execution trace expressed over basic blocks—and therefore only pays for blocks that actually execute on a given input. CoBBL's new proof-specific execution model makes it possible for the compiler to optimize how state travels from one basic block to another. In this execution model, there are unlimited, cheap local registers in a given basic block; to pass state between blocks, it's possible to use either non-local registers or a stack that captures the scoping behavior of the high-level program. The compiler determines, for a given program, the optimal balance of register and stack use.

Realizing this execution model, compiler, and poof system required three key components. First, we design program-analysis machinery that lets CoBBL balance the size of explicitly passed state (i.e., non-local registers) against the number of memory reads and writes (i.e., stack usage). Second, we design a concretely efficient *scope stack* implementing the semantics required for (non-heap) program state. Because our design is precisely tailored to stack semantics, it avoids the overhead of a general-purpose RAM abstraction; because it is optimized for a specific program, it imposes significantly less overhead than a generic call stack.

The third component is a basic-block representation of the program and a corresponding proof-system backend that jointly minimize costs. CoBBL's prover costs scale both with the number of distinct basic blocks in the program's execution trace and with the maximum size of any block. As one example of minimizing this cost, we design compiler optimizations that allow CoBBL to control both the number and the size of the generated basic blocks.

The result of our work is a co-designed execution model, compiler, and proof system, CoBBL, that effectively delivers the upsides of both direct translation and CPU emulation. Our results show that CoBBL outperforms a state-of-the-art direct translator [62] by 1–30× on compile time and 26–350× on prover time. CoBBL's prover performance is also 1.1–1.8× faster than a state-of-the-art CPU emulator [9] on programs involving the CPU's native instructions, and 100× faster on computations involving the proof system's native field arithmetic.

In the rest of this paper, we give background on SNARKs and related work on direct translation and CPU emulation (§2). We then describe the design of CoBBL's compiler (§3.1) and proof back-end (§3.2), then evaluate against state-of-the-art prior work (§4). Finally, we demonstrate that CoBBL handles a complex, real-world application that should *in principle* be well suited to proof systems, but which prior systems simply cannot handle (§5). We close with a discussion of future directions (§6).

2. Background and related work

This section reviews the two predominant approaches to constructing SNARKs, *direct translation* and *CPU emulation*, and their advantages and disadvantages. It also describes in detail the most closely related prior approaches.

2.1. SNARK overview

SNARK systems [14, 17, 21, 36, 37, 41, 44, 52, 68] enable the following arrangement: Given a fast but potentially malicious prover $\mathcal{P}$, a slow but trusted verifier $\mathcal{V}$, a program Ψ, and some input to the program x, $\mathcal{V}$ outsources the computation $\Psi(x)$ to $\mathcal{P}$ and receives the output y along with a short proof π that (probabilistically) convinces $\mathcal{V}$ that $y = \Psi(x)$. A SNARK satisfies the following properties:[1]

1) *Completeness*: if $y = \Psi(x)$, an honest $\mathcal{P}$ can convince $\mathcal{V}$ except with negligible probability.

[1]We emphasize that this work *does not* attempt to give a zero-knowledge property; we discuss in Section 6.

2) *Soundness*: if $y' \neq \Psi(x')$, a dishonest $\mathcal{P}$ cannot convince $\mathcal{V}$ except with negligible probability.[2]
3) *Succinctness*: $\mathcal{V}$'s cost to verify π is sublinear in the work required to compute $\Psi(x)$.

Essentially all SNARK systems require transforming the computation Ψ into a set of constraints $\mathcal{C}$, defined over some finite field $\mathbb{F}$, whose satisfiability is tantamount to correct execution of Ψ. Slightly more formally, we say that the constraints $\mathcal{C}$ are over vectors $X \in \mathbb{F}^{|X|}$, $Y \in \mathbb{F}^{|Y|}$, and $W \in \mathbb{F}^{|W|}$, where X and Y are the distinguished input and output variables, respectively, and the remaining variables W are called the *witness* (intuitively, the vector W corresponds to the intermediate states of the computation Ψ). We say that $\mathcal{C}$ *implements* Ψ just when $y = \Psi(x) \iff \forall x \in \mathbb{F}^{|X|}, \exists y \in \mathbb{F}^{|Y|}, w \in \mathbb{F}^{|W|} : \mathcal{C}_{X=x,Y=y,W=w}$ (i.e., $\mathcal{C}$ where X takes the value x, Y takes y, and W takes w) is satisfied. Put another way: when $y = \Psi(x)$, there exists an assignment to W satisfying the constraints $\mathcal{C}_{X=x,Y=y}$; if $y' \neq \Psi(x')$, $\mathcal{C}_{X=x',Y=y'}$ is unsatisfiable.

For $\mathcal{C}$, Ψ, and x, an honest $\mathcal{P}$ convinces $\mathcal{V}$ as follows. First, it computes $y = \Psi(x)$. Next, it uses the execution transcript of Ψ to compute an assignment to W that satisfies $\mathcal{C}_{X=x,Y=y}$. It then invokes some cryptographic and complexity-theoretic machinery to produce a proof string π. Finally, $\mathcal{P}$ sends y and π to $\mathcal{V}$, which uses corresponding cryptographic and complexity-theoretic machinery to verify the proof. If verification succeeds, $\mathcal{V}$ is convinced that $\mathcal{P}$ knows w such that $\mathcal{C}_{X=x,Y=y,W=w}$ is satisfied, and therefore that $y = \Psi(x)$.

The precise complexity-theoretic and cryptographic machinery used varies among SNARKs (e.g., [25, 34, 45, 65, 72]). Regardless of the specifics of this machinery, $\mathcal{P}$'s cost (and, to a lesser degree, $\mathcal{V}$'s) depends crucially on the size of the constraint set, $|\mathcal{C}|$. The quest to let programmers easily construct $\mathcal{C}$ for a desired Ψ while minimizing the proving and verifying cost has yielded two general classes of SNARK systems: direct translation and CPU emulation. We now discuss each in more detail.

Direct translation. Several lines of work, including [20, 28, 50, 62, 63, 66, 67, 70], build a constraint set using a special-purpose compiler that turns each step of Ψ into a corresponding set of constraints. As a very simple example, addition and multiplication in Ψ might be translated into corresponding arithmetic expressions in $\mathcal{C}$.

The primary advantage of direct translation is that it creates program-specific constraints, meaning it can take advantage of *both* the program's structure and the constraint-satisfaction setting to find optimizations. As a simple example, consider the computation $\Psi(x) = x^{-1} \in \mathbb{F}$. While *computing* the multiplicative inverse of x has cost logarithmic in $|\mathbb{F}|$, e.g., using Euclid's algorithm, $\mathcal{C}$ can be exceedingly simple: $y \cdot x = 1 \in \mathbb{F}$ is satisfiable iff x is invertible and y is its inverse. The key observation here is that variables in $\mathcal{C}$ other than X (which is fixed by $\mathcal{V}$) are *existentially quantified*, meaning that $\mathcal{C}$ need only encode the *correctness condition* for y and w—and not the method of computing the satisfying assignment. As the multiplicative-inverse example shows, optimizations that use nondeterminism can achieve huge savings, both concretely and asymptotically (e.g., [47]).

A well-known downside of all existing direct-translation approaches stems from the fact that $\mathcal{C}$ must be fixed at compile time, meaning that it must encode *all possible execution paths* of Ψ. As a result, $\mathcal{C}$ must *flatten* all control flow: it must contain constraints for both branches of every conditional statement, all loops in Ψ must be unrolled to a static depth, and all function calls must be inlined. This means that $\mathcal{P}$ pays the cost of both branches no matter which is taken, and likewise it pays the maximum cost for all loops no matter how many loop iterations are actually required in a given execution.

A more insidious result of control-flow flattening is that the cost to directly translate Ψ to $\mathcal{C}$ is (at least) proportional to *the running time* of Ψ. (In contrast, for compilers targeting CPUs, compilation time is proportional to the size of the *program representation*—which is almost always much less than the runtime because, for example, loops are not generally unrolled and nontrivial functions are rarely inlined.) Making matters worse, compiler optimizations are often super-linear in the size of the code unit being optimized, and many optimizations in prior work are applied to the entire constraint set. This means that existing works using direct translation have extremely high compilation costs, both concretely and asymptotically.

Prior work attempts to press advantages or sidestep issues by producing constraints further tailored to each program and proof system. As examples, Buffet [70] automatically flattens complex, nested loops, avoiding multiplicative blow-up that would otherwise result from unrolling nested loops to static bounds; and Giraffe [71] uses program analysis to "squash" iterative loops into data-parallel circuits, capitalizing on its proof back-end's efficiency on computations of this form. Unfortunately, no prior work resolves the problem that constraints must be fixed at compile time.

CPU emulation. Several other lines of work [9, 10, 12, 13, 18, 22, 39, 58, 64, 75, 77] take a different approach. Rather than directly encoding Ψ, $\mathcal{C}$ encodes the fetch-decode-execute loop of a general-purpose CPU. The computation Ψ is compiled to a sequence of instructions for this CPU, and the inputs for $\mathcal{C}$ encode both this sequence of instructions and the arguments to Ψ being evaluated. Put another way: $\mathcal{C}$ implements a CPU emulator whose input is the program to be run, i.e., $\Psi(x)$.

Originally envisaged as a way to create $\mathcal{C}$ once and for all, independent of Ψ [11–13, 18, 58], systems based on CPU emulation have seen increasing interest because they sidestep some of the thorniest issues with direct translation. Because $\mathcal{C}$ encodes the fetch-decode-execute loop of the emulated CPU, there is no need to flatten control flow.

[2] A SNARK actually satisfies a stronger property, *knowledge soundness*, which intuitively means that when a prover produces a convincing proof, $y = \Psi(x)$ *and* $\mathcal{P}$ actually holds a valid execution transcript. This is formalized via the existence of an efficient *extractor* algorithm $\mathcal{E}$ that, given oracle (possibly rewinding) access to a convincing prover $\mathcal{P}$, outputs the execution transcript except with negligible probability.

Instead, this flattening is applied to the CPU's fetch-decode-execute loop itself: $\mathcal{C}$ encodes the execution of a sequence of CPU instructions. This means that $\mathcal{P}$ proves only the correctness of the instruction sequence actually executed to evaluate $\Psi(x)$ (plus, potentially, some no-ops).

In addition, if the CPU being emulated already has compiler support, there is no need to build a special-purpose compiler to translate Ψ into a format suitable for proving. (Even if the CPU's instruction set is unusual, LLVM [54] and related projects make it relatively easy to graft a custom instruction set onto a mainstream compiler.) This means that compilers for CPU-emulator–based proofs have the same performance characteristics as standard compilers.

A significant downside of CPU emulators is that they use random-access memory to store and manage program state.[3] This means that a significant fraction of the emulated CPU cycles do not do useful work; rather, they are spent on memory operations that move data to and from registers and track control flow. This behavior is a consequence of the way that modern CPUs are designed to accommodate compilation, namely, with support for efficient use of a call stack [32]. For a real CPU, the call stack is efficient because it often resides in cache, making accesses very cheap. CPU emulators, in contrast, pay the full cost of the (expensive) memory abstraction (plus an emulated CPU cycle, e.g., for a load, store, push, or pop) for every stack operation.

Another drawback of CPU emulators is that they give up the opportunity for program-specific optimization *at the constraint level*. As a result, although $\mathcal{C}$'s implementation of the CPU's fetch-decode-execute loop itself almost always takes advantage of existential-quantification–based optimizations, the implementation of Ψ itself cannot do so. In the case of the multiplicative-inverse example mentioned above, this means that what would be one constraint for a direct-translation–based system might instead require thousands of CPU instructions (each comprising tens or hundreds of constraints) to implement (say) Euclid's algorithm. A further consequence is that, even if Ψ uses operations in the same field $\mathbb{F}$ used for $\mathcal{C}$, CPU-emulation–based systems generally encode those field operations as CPU instructions (i.e., multi-precision arithmetic), easily increasing the cost of such operations by several orders of magnitude [70, §5].[4]

One partial response to the above problem is to extend the definition of the CPU to include functionalities that are tailored to a specific computation [2, 53, 64]. Unfortunately, these custom instructions are generally hand built, meaning they are restricted to the most frequently used primitives. In addition, almost all CPU-emulation–based systems have the explicit or implicit goal of using an essentially off-the-shelf compiler, which therefore does not know how to select and emit custom instructions—so these instructions must instead be manually invoked (e.g., via compiler intrinsics). In other words, only experts who know how to tune the proof machinery can create new functionalities, and those functionalities can only give improved performance if the programmer rewrites their program.

[3] The program being executed is also stored in memory. It is often possible to use a less expensive read-only memory abstraction for code; this reduces but does not completely eliminate the cost of fetching instructions since *every* cycle requires a fetch.

[4] As one example: a standard optimization when Ψ involves elliptic-curve operations is to choose the constraint field $\mathbb{F}$ coinciding with the elliptic curve's base field. This optimization makes such operations extremely cheap for direct-translation–based systems, but it does not apply to most CPU-emulator–based systems.

Closely related prior work: vRAM. vRAM [75] is a CPU emulator that dynamically generates the constraint set $\mathcal{C}$ from the exact sequence of instructions executed by the program Ψ on a *specific* input. Since vRAM generates $\mathcal{C}$ on the fly, it proves constraints tailored to the instructions actually executed, rather than a constraint set encoding a generic CPU transition function—the approach that was, up to that time, state of the art [11–13]. (More recent approaches, e.g., [9, 40], achieve the same "pay-for-what-you-use" approach as vRAM in other ways.)

Given a trace of the instructions executed and all intermediate values, $\mathcal{P}$ proves to $\mathcal{V}$ that: (i) every instruction is executed correctly, (ii) the values of CPU registers are consistent between successive instructions, and (iii) the state of memory is coherent throughout the execution. To do so, $\mathcal{P}$ expresses the input-output register state of every executed instruction in three orderings:

- The *execution ordering* lists register states in the order instructions were executed. $\mathcal{P}$ proves statement (ii) by showing that the output state of every instruction matches the input state of the next one.
- The *instruction ordering* groups all input-output register states by the type of instruction. $\mathcal{P}$ proves, for each instruction type i and its constraint representation $\mathcal{C}_i$, that all instructions purportedly of type i satisfy the constraints $\mathcal{C}_i$, which proves statement (i).
- The *memory ordering* sorts all memory accesses by address, as in the sorting-based RAM abstraction described in Section 2.2. $\mathcal{P}$ proves statement (iii) using that abstraction's memory coherence checks.

Finally, $\mathcal{P}$ convinces $\mathcal{V}$ that the above three orderings of register states are permutations of each other using a classic multiset equality check (see §3.2, "Trace compatibility").

Looking ahead, vRAM's approach directly inspires ours. The most significant difference is that CoBBl works directly on a constraint representation of the program, rather than on CPU instructions. This introduces several challenges (discussed in the Introduction), but means that CoBBl can take advantage of per-program constraint- and execution-model optimizations, whereas vRAM cannot.

2.2. Primitives

Sorting-based RAM abstraction. Ben-Sasson et al. [11] introduce a now-standard approach to encoding random-access memory semantics as a constraint satisfaction problem; many follow-up works refine [12, 13, 50, 70]. In this approach, the prover records memory operations executed during the computation in a transcript, as tuples $(\texttt{addr}, \texttt{val}, \texttt{ts}, \texttt{ls})$. Here, `addr` is the address, `val` is the data read or written, `ts` is a timestamp that starts from 0

and is incremented after each memory operation, and `ls` indicates whether the operation is a load or store.

To prove correctness of a sequence of memory operations, the prover first produces a *memory-ordered* version of the transcript, i.e., one sorted by `addr`, breaking ties with `ts`. Next, it proves that each sequential pair of entries in the memory-ordered transcript satisfies the memory coherence conditions, i.e., every executed load returns the previously stored value (or a default value if no store occurred). Finally, it proves that the memory-ordered transcript is properly sorted, i.e., that it is a permutation of the execution-ordered transcript and that sequential pairs in the memory-ordered transcript are ordered correctly.

In sum, the cost of this approach is the cost of a permutation check (which has linear cost via a public-coin interactive protocol) plus the cost of the memory coherence conditions, whose leading cost is in checking that the entries are properly sorted. In particular, evaluating the inequalities $\texttt{addr}_i \leq \texttt{addr}_{i+1}$ and $\texttt{ts}_i \leq \texttt{ts}_{i+1}$ imposes overheads logarithmic in the size of the address space and in the number of accesses to memory, respectively.

Sequential, write-once RAM. Recently, Goldberg et al. [40] described a variant of the generic read-write RAM construction that reduces costs but imposes two additional requirements: first, each address in memory can be written exactly once; and second, the set of addresses that are accessed must form a contiguous sequence. These conditions eliminate both of the inequalities responsible for the logarithmic overheads in the general-purpose construction. Specifically, since the sequence of addresses is contiguous, the $\texttt{addr}_i \leq \texttt{addr}_{i+1}$ check can be replaced with $\texttt{addr}_i = \texttt{addr}_{i+1} \vee \texttt{addr}_i + 1 = \texttt{addr}_{i+1}$, which costs one constraint. Likewise, the write-once requirement means that every operation on a given address will have the same `val`, entirely avoiding the use of the `ts` and `ls` fields.

Polynomial commitment schemes (informal). A *polynomial commitment scheme* [48] allows a committer to bind itself to a specified polynomial by sending a *commitment* string to a recipient. Later, the committer can *evaluate* the polynomial at a point specified by the recipient, and produce the evaluation plus a proof that the evaluation is consistent with the commitment string. Informally, a polynomial commitment scheme ensures that the committed polynomial has bounded degree, that the committer cannot equivocate about the committed polynomial or a claimed evaluation, and that the commitment and proofs do not reveal anything about the polynomial other than the evaluations themselves. A polynomial commitment scheme is *extractable* if there exists an efficient algorithm $\mathcal{E}$ that, given oracle (possibly rewinding) access to a convincing prover $\mathcal{P}$, outputs the committed polynomial except with negligible probability.

Polynomial commitment schemes for different types of polynomials exist; we are primarily concerned with schemes that support univariate and/or multilinear polynomials. In particular, the Hyrax polynomial commitment scheme [72] and related schemes (e.g., [42]) are flexible: the committed polynomial can be treated either as a univariate polynomial (in the standard monomial basis) or as a multilinear polynomial (defined by evaluations on the boolean hypercube).

3. Design

CoBBL combines best aspects of the direct-translation and CPU-emulation approaches (§2.1). Specifically: it takes advantage of low-level program-specific optimizations and leverages nondeterminism (i.e., check the result rather than computing it), like a direct translator. At the same time, like a CPU emulator, it avoids paying for branches that are not taken and loop iterations that are not needed when proving correct execution for a given input. And CoBBL's compilation time depends on the size of a computation's description, not on its running time. (Unlike a CPU emulator, however, CoBBL cannot use an off-the-shelf compiler because it does not target a CPU abstraction.)

Most importantly, unlike a CPU emulator, CoBBL avoids paying for costly memory operations associated with managing program state. In particular: CoBBL is not restricted to a fixed number of CPU registers, meaning that it does not pay to shuffle local variables between the stack frame and the registers. Likewise, rather than a generic calling convention, CoBBL's compiler optimizes the way that non-local variables are passed through the program. To further reduce costs associated with non-local variables, CoBBL uses an optimized stack abstraction.

Our starting point in designing CoBBL is vRAM (§2.1). Like vRAM, CoBBL dynamically generates constraints corresponding to the execution being proved—meaning that it does not contain extraneous constraints for untaken branches or dummy loop iterations. Unlike vRAM, however, CoBBL does not prove execution of a sequence of CPU instructions. Instead, the CoBBL compiler takes a program Ψ as input and breaks it into a control flow graph (CFG) of *basic blocks*. Then, it applies program analyses and optimizations that are tailored to the proof-systems context, and finally it emits constraints for each basic block. To generate a proof on a given input, the CoBBL prover executes the program and records the resulting sequence of basic blocks and intermediate values, producing an *execution trace*, then proves that this trace contains the correct sequence of blocks and that each block was correctly executed.

Input language. CoBBL's input language is an extension of Z# [62]. Z# supports integer, boolean, and field operations, ternary operators, statically-bounded for-loops, structs, and arrays. Our extension adds support for if-else statements and *unbounded* while- and for-loops. CoBBL builds on CirC [62], so it is relatively easy to extend to other input languages. We stress that CoBBL's techniques are *not* specific to the input language.

Basic blocks. CoBBL compiles input programs into a CFG of basic blocks expressed in CoBBL IR. A basic block consists of a label (b), sets of input registers $(\text{reg}_{i0}, \text{reg}_{i1}, \ldots)$ and output registers $(\text{reg}_{o0}, \text{reg}_{o1}, \ldots)$, and a sequence of IR instructions. There are five special registers in CoBBL IR: `%BN` stores the block label, `%RP` stores the label of

the block to return to after a function call, %SP and %BP define the *scope stack* (analogous to a call stack; §3.1), and %TS records the number of memory accesses (for the memory coherence check, §3.2). A block's input and output register sets include the special registers plus general-purpose registers used to pass values between blocks.[5]

The first instruction of each block asserts that %BN matches the block's label. The last instruction updates %BN to the label of the next block. This instruction either: sets %BN to %RP when control returns from a function call; is an unconditional jump that assigns %BN a constant value; or is a conditional jump that assigns %BN the result of a ternary operation. The remaining instructions include arithmetic and logical expressions, assignment to registers, assertions, ternary operators, memory operations (i.e., load and store), and pushes to and pops from the scope stack (§3.1).

Executing a program. Given a set of program inputs, $\mathcal{P}$ begins executing at a distinguished starting block. Initially, memory and the scope stack are both empty. For each block, $\mathcal{P}$ executes the sequence of instructions subject to the values of the input registers, scope stack, and memory (as a special case, the initial block's input registers are just the program inputs). As described above, the final instruction of every block updates %BN to indicate which block is executed next. Values in the registers at the end of a block's execution become that block's output registers, and the successor block's input registers. $\mathcal{P}$ continues executing blocks until the program terminates on a distinguished halting block. $\mathcal{P}$ records all block inputs and outputs, intermediate values, and memory operations for use during proving.

Proving correct execution. To convince the verifier that it correctly executed the program on the specified inputs, the prover generates a succinct non-interactive proof using an efficient protocol that is tailored to CoBBL's program representation and execution trace. We describe the statement being proved and the protocol in Section 3.2. Before doing so, in Section 3.1 we detail the design of CoBBL's compiler and its proof-specific optimizations.

3.1. Creating and optimizing basic blocks

CoBBL initially generates a graph of basic blocks where each scope change (e.g., a loop body, or the consequent or alternative of an if-else) results in a new block, and where functions introduce a new series of blocks. CoBBL then performs standard optimizations like function inlining and dead code elimination, in addition to optimization passes specifically designed to make proving and verifying cheaper; this section describes the latter.

Minimizing the number of blocks and block size. The goal of this optimization pass is to minimize the number of blocks in the program while making sure that no one block becomes too large. CoBBL achieves this by merging smaller blocks together, respecting a developer-selected bound on MAX_BLOCK_SIZE. The first step is size estimation: CoBBL estimates the number of constraints $\mathcal{C}_b$ that it would take to express each basic block b. Then, it computes $|\mathcal{C}_{\max}|$, the maximum size of any merged block; $|\mathcal{C}_{\max}|$ is the maximum of MAX_BLOCK_SIZE and the largest block size in the program. Finally, it uses this bound to iteratively merge blocks until those blocks would exceed $\mathcal{C}_{\max}$ in size.

The block merging algorithm uses existing techniques [70] to flatten control flow branches (e.g., from a conditional statement or a loop) into a single series of instructions and conditional selects. This means that merged blocks require $\mathcal{P}$ and $\mathcal{V}$ to pay to execute both branches of a conditional statement, where previously, they would only pay for one branch. The tradeoff between reducing the number of blocks and increasing the cost of unexecuted branches is the main factor to consider when choosing MAX_BLOCK_SIZE.

To merge blocks, CoBBL keeps track of $\mathcal{M}_b$, the estimated size of the merged block starting at block b. If block b does not initiate a conditional jump, CoBBL sets $\mathcal{M}_b$ to $|\mathcal{C}_b|$; the remaining conditions guarantee that this block will be merged into a predecessor block if possible. If block b initiates a function call, merging is undesirable—any uninlined function has multiple callsites. To denote this, CoBBL sets $\mathcal{M}_b$ to $\perp$, which prevents merging. Similarly, if block b is the head of a loop of *unknown* bound, CoBBL does not know how many times to unroll the loop, so it sets $\mathcal{M}_b = \perp$ and leaves the loop unchanged. Finally, if block b is the head of a conditional jump with branch targets ℓ and r, and join point block t, CoBBL recursively computes $\mathcal{M}_\ell$, $\mathcal{M}_r$, and $\mathcal{M}_t$; merging all of these blocks together would yield a block of size $\mathcal{M}_b = |\mathcal{C}_b| + \mathcal{M}_\ell + \mathcal{M}_r + \mathcal{M}_t$. If $\mathcal{M}_i$ does not exceed $|\mathcal{C}_{\max}|$, CoBBL merges blocks b, ℓ, r, and t; otherwise, it leaves them alone. Unrolling a loop of static bound n is similar, except $\mathcal{M}_b = |\mathcal{C}_b| + n \cdot \mathcal{M}_{body} + \mathcal{M}_t$.

Balancing cross-block register passing with scope-stack operations. This pass optimizes the maximum number of registers passed between any two blocks in the program. In CoBBL, the alternative to passing a value from one block to the next in a register is to push the value onto the *scope stack*, which behaves roughly like a traditional call stack. In contrast to a call stack, however, CoBBL's scope stack only stores data that is persistent from one block (or scope) to the next. In particular, block-local variables never appear on the scope stack: they are just wires in $\mathcal{C}_b$.[6] The goal of this optimization is thus to balance the number of registers passed between blocks against the number of scope-stack operations. We now describe both the optimization pass and CoBBL's scope stack implementation, which uses relatively cheap sequential, write-once memory (§2.2).

The spilling algorithm first bounds the maximum number of registers passed between any two blocks as $\mathsf{o}_{\text{width}}$, the

[5] We refer to %BN, %RP, %SP, %BP, and %TS as registers for clarity of exposition. %BN, %RP, and %TS are more properly regarded as global variables; in principle they could be stored on the scope stack.

[6] As an optimization, CoBBL also allows the programmer to initialize constant arrays on the scope stack. These arrays offer indexed access (e.g., access at an offset that depends on a computation's input) at significantly lower cost than heap (i.e., read/write) storage.

maximum number, over all blocks, of variables that are both in-scope and alive after the end of the block. (Observe that these values *must* be passed to the next block.)

For each block CoBBl will spill other non-local variables (i.e., those that are live but *not* in scope, e.g., variables from an enclosing scope that are shadowed by another live value), stopping once the number of block output variables is no more than $\mathtt{o}_{\text{width}}$. In other words, if there are $\mathtt{o}_b$ live variables in total at the end of block b, CoBBl spills at least $s_b = \mathtt{o}_b - \mathtt{o}_{\text{width}}$ of them onto the stack.

To choose which variables are spilled in blocks where s_b is nonzero, CoBBl uses the CFG to select a live, shadowed variable whose shadowing variable is in the outermost scope. (For this purpose, CoBBl's CFG captures scope information.) In particular, if the shadowed variable is not already marked spilled, CoBBl marks it for spilling at the entry to the outermost scope in which it is shadowed; in effect, this variable is spilled in *every* basic block, not just the one being analyzed. The compiler continues this process until at least s_b variables are spilled in each block b.

The scope stack. As previously mentioned, CoBBl defines the scope stack using two special registers: $\%\mathtt{SP}$ points to the top of the stack, $\%\mathtt{BP}$ to the start of the current stack frame, much as in a standard call stack. The scope stack differs from a call stack because it is designed to use a sequential, write-once memory (§2.2), which is less expensive in the proofs context. In effect, CoBBl's scope stack implements push and pop using an append-only log.

At the beginning of the program, $\%\mathtt{SP}$ and $\%\mathtt{BP}$ are 0. When the program enters a new scope (i.e., when the scope depth increases), it spills the s variables that were marked by the spilling algorithm. To do this, the compiler emits instructions to: allocate a new stack frame of size $s+1$ in the write-once memory; write the current value of $\%\mathtt{BP}$ into the first allocated entry (which is pointed to by $\%\mathtt{SP}$); set $\%\mathtt{BP} = \%\mathtt{SP}$; store the spilled variables into the remaining s entries; and set $\%\mathtt{SP} = \%\mathtt{SP} + s + 1$. When $s = 0$, the compiler does not emit these instructions.

When the program exits a scope (i.e., when scope depth decreases), it pops off the stack the variables that were spilled at scope entry. For this purpose, the CoBBl compiler emits instructions to load s values from the stack, starting from address $\%\mathtt{BP} + 1$, then to load the prior value of $\%\mathtt{BP}$, which is stored at $\%\mathtt{BP}$. Crucially, in this approach CoBBl *never* overwrites previously written stack values: instead, it allocates fresh space for each stack frame, meaning that the value of $\%\mathtt{SP}$ never decreases. For correctness of the scope-stack approach, CoBBl requires every scope entry to have a matching scope exit. This requirement, while mild, rules out certain programs that, e.g., dynamically generate code or abuse function pointers; see Section 6.

3.2. Proving correct execution

An execution trace corresponds to a correct execution of the program on the specified inputs iff the following conditions hold:

1) the execution trace begins at the starting block, which takes the specified inputs, and finishes after ℓ steps on the halting block, which produces the claimed outputs;
2) the trace contains the correct sequence of blocks;
3) register values are passed correctly between blocks;
4) each block in the trace is executed correctly; and
5) both RAM and the scope stack behave correctly.

To establish these conditions, $\mathcal{P}$ uses three related representations of the program's execution:

- *Execution trace*: as described above, this trace encodes a sequence of basic blocks and their input and output register values, in the order they were executed.
- *Block$_b$ traces*: for each block label b corresponding to a block in the execution trace, the Block$_b$ trace encodes the input and output register values for all copies of block b in the execution trace, plus all witness values required to prove satisfaction of block constraints $\mathcal{C}_b$ for those blocks.
- *Memory traces*, one for the heap (i.e., the general-purpose RAM) and one for the stack (i.e., the sequential, write-once RAM). Each trace comprises a sequence of tuples representing the memory operations on the respective memory during the program's execution, sorted by the memory address accessed (§2.2).

Protocol overview. Mechanically, $\mathcal{P}$ sends $\mathcal{V}$ polynomial commitments (§2.2) encoding each of these traces (details are below) and proves that the above conditions hold with respect to the committed values. In particular:

- $\mathcal{P}$ establishes condition 1 with respect to the execution trace by opening the parts of the trace pertaining to program inputs and outputs. In particular, $\mathcal{V}$ checks that the inputs and outputs are correct and that in the ℓ^{th} set of register values $\%\mathtt{BN}$ gets a distinguished HALT value.
- $\mathcal{P}$ establishes condition 3 with respect to the execution trace by proving that, for all inputs and outputs encoded in the trace, block j's outputs equal block $j+1$'s inputs. Details are given in "Register consistency," below.
- $\mathcal{P}$ establishes condition 4 with respect to the Block$_b$ traces using data-parallel Spartan proofs [65, 68, 69, 71] that the values encoded in those traces comprise satisfying assignments for each constraint set $\mathcal{C}_b$.
- $\mathcal{P}$ proves that the memory traces satisfy the memory coherence checks (§2.2) using data-parallel Spartan proofs. Memory operations are also encoded in the satisfying assignments to $\mathcal{C}_b$; the proof establishing condition 4 ensures correct operations in the Block$_b$ traces. The trace compatibility conditions described below ensure agreement between the memory and Block$_b$ traces. Taken together, these establish condition 5.
- Condition 2 is a consequence of the $\%\mathtt{BN}$ register, the way $\mathcal{C}_b$s are constructed, and conditions 3 and 4. Specifically, when block b_j transfers control to block b_{j+1}, $\mathcal{C}_{b_j}$ sets the output $\%\mathtt{BN}$ value to the label of the successor block, b_{j+1}; condition 3 ensures that the successor block's input $\%\mathtt{BN}$ value matches the value set by the predecessor; and $\mathcal{C}_{b_{j+1}}$ asserts that the input $\%\mathtt{BN}$ value equals b_{j+1}.

Finally, $\mathcal{P}$ must establish that all of the execution, Block$_b$, and memory traces are *compatible*, i.e., that they

correspond to the same computation. In particular, the Block_b traces must comprise a partitioning of the execution trace by block label, meaning that each register input/output pair in the execution trace appears in exactly one of the Block_b traces; and each RAM operation in the Block_b traces must appear in the heap's memory trace, and likewise for stack operations and the stack's memory trace.

We now describe the sub-protocols that $\mathcal{P}$ uses to establish all of the conditions described above.

Trace compatibility. The trace compatibility conditions boil down to showing that a set committed in the Block_b traces is equal to a set committed in the execution trace or in one of the memory traces. $\mathcal{P}$ does this using a classical proof of multiset equality that Blum and Kannan ascribe to Lipton [19]: interpret each set as a polynomial over a large field with roots defined by the elements of the set, i.e., for $A = \{a_0, a_1, \ldots\}$, $P_A(t) = \prod_i (\tau - a_i)$, and analogously for B and $P_B(t)$. Then if $P_A(\tau) = P_B(\tau)$ for random τ, A and B are equal with high probability.

In our case, the sets being compared are sets of *tuples*, i.e., of registers $(\text{reg}_{i0}, \text{reg}_{i1}, \ldots, \text{reg}_{o1}, \text{reg}_{o2}, \ldots)$ or of memory-operation arguments $(\texttt{addr}, \texttt{val}, \texttt{ts}, \texttt{ls})$. Ordering within each tuple is ensured by *fingerprinting*, i.e., treating each tuple's entries as the coefficients of a univariate polynomial $\text{tup}(r)$ over a large field, evaluating at a random point, and proving equality of sets of fingerprints.

In more detail: after $\mathcal{P}$ commits to the execution, Block_b, and memory traces, $\mathcal{V}$ samples and sends τ and ρ. $\mathcal{P}$ then sends commitments to a set of partial-products vectors $T = \Gamma_0, \gamma_0, \Gamma_1, \gamma_1, \ldots$, specifically, one vector comprehending the registers in the execution trace, one vector comprehending the memory operations in each memory trace, and three vectors for each Block_b trace (one for stack memory, one for heap memory, and one for registers). For each such commitment, $\mathcal{P}$ also sends a data-parallel Spartan proof [65, 68, 71] that the committed γ_i values are correctly computed, where each data-parallel sub-computation establishes that $\gamma_i = \tau - \text{tup}_i(\rho)$ for $\text{tup}_i(r)$ computed over entries in the execution, Block_b, or memory trace corresponding to the entry in the committed partial-products vector. $\mathcal{P}$ then uses grand-product proofs (Appx. A.2) to help $\mathcal{V}$ evaluate $\prod_i (\tau - \text{tup}_i(\rho))$ for each partial-products vector.

Finally, $\mathcal{V}$ checks that the product of the Block_b register grand products equals the execution trace's grand product, the product of the Block_b heap grand products equals the heap memory trace's grand product, and the product of the Block_b stack grand products equals the stack memory trace's grand product. If $\mathcal{V}$ accepts, the execution and Block_b traces have compatible register values, and the Block_b and memory traces have compatible memory operations except with negligible probability over $\mathcal{V}$'s random choices.

Register consistency. To prove that block j's outputs equal block $j+1$'s inputs in the execution trace (condition 3), $\mathcal{P}$ uses fingerprinting (defined above), a *shift proof* (Appx. A.1), and Spartan [65]. While slightly indirect, our approach is designed to enable the use of *data-parallel* [68, 71] Spartan proofs and is thus very efficient.

In more detail: $\mathcal{P}$ commits to a vector of fingerprints, $R = \mathsf{fp}_{i,0}, \mathsf{fp}_{o,0}, \mathsf{fp}_{i,1}, \mathsf{fp}_{o,1}, \ldots, \mathsf{fp}_{i,n}, \mathsf{fp}_{o,n}$, where each fingerprint corresponds to either a tuple of input-register values, $\mathsf{fp}_{i,\cdot}$ or a tuple of output-register values, $\mathsf{fp}_{o,\cdot}$. (In contrast, the fingerprints used for trace compatibility represent tuples containing *both* input and output register values.) $\mathcal{P}$ also commits to the same vector purportedly shifted two entries to the left, i.e., $S = \mathsf{fp}_{i,1}, \mathsf{fp}_{o,1}, \mathsf{fp}_{i,2}, \mathsf{fp}_{o,2}, \ldots, \mathsf{fp}_{i,n}, \mathsf{fp}_{o,n}, 0, 0$. $\mathcal{P}$ then proves that the two commitments have the required shift relation using the protocol described in Appendix A.1.

Finally, $\mathcal{P}$ produces a data-parallel Spartan proof over the commitments to the R and S vectors and the committed execution trace. Each of the data-parallel sub-computations in this proof checks that the corresponding R entry was correctly computed (by fingerprinting the input and output register tuples contained in the execution trace) and that the committed fingerprints demonstrate register consistency (i.e., that $\mathsf{fp}_{o,0}$ in the R commitment is equal to $\mathsf{fp}_{i,1}$ in the S commitment, and so on). Using both the R and S commitments enables data parallelism; the shift proof ensures that checking $\mathsf{fp}_{o,0}$ in R against $\mathsf{fp}_{i,1}$ in S is sound.

Summary and asymptotics. We now describe the full public-coin interactive protocol; it can be made non-interactive in the random oracle model via the Fiat-Shamir heuristic.

1) $\mathcal{P}$ commits to the execution, Block_b, and memory traces using the Hyrax polynomial commitment scheme, then sends the commitments to $\mathcal{V}$.
2) $\mathcal{V}$ samples and sends τ and ρ.
3) $\mathcal{P}$ uses τ and ρ to compute the partial-products vectors for the trace compatibility proofs and the fingerprint vectors for the register consistency proofs. $\mathcal{P}$ commits to the vectors and sends the commitments to $\mathcal{V}$.
4) $\mathcal{V}$ samples and sends ζ for the shift proofs.
5) $\mathcal{P}$ computes data-parallel Spartan proofs for trace compatibility, register consistency (condition 3), grand products, memory coherence (condition 5), and satisfaction of all $\mathcal{C}_b$ (condition 4). $\mathcal{P}$ then generates all polynomial-commitment opening proofs for the Spartan and shift proofs. Finally, $\mathcal{P}$ opens relevant commitments at the required points for the shift proofs and condition 1. $\mathcal{P}$ sends all proofs and openings to $\mathcal{V}$.
6) $\mathcal{V}$ verifies all proofs and correct input/output binding.

$\mathcal{P}$'s costs in the above protocol is linear in the runtime of the program: all commitments are at most linear in the runtime; Hyrax's commitment cost is linear in the size of the value committed; and the costs to generate the data-parallel Spartan proofs are linear, except that the memory consistency proofs for the heap (i.e., general-purpose read/write memory) impose an overhead that is logarithmic in the total number of accesses to heap memory (§2.2). We emphasize that this cost profile is better than existing CPU-emulation–based systems, which use general-purpose read-write memory for both the heap and the stack.

$\mathcal{V}$'s costs in the above protocol are linear in the number of inputs and outputs (this is fundamental: $\mathcal{V}$ must check that

inputs and outputs are correctly bound) and in the number of distinct block types executed by the program (opening the Block_b commitments and verifying the Spartan proofs for $\mathcal{C}_b$). Because of the Hyrax polynomial commitment, $\mathcal{V}$'s costs scale with the square-root of the program's runtime.

Security argument in brief. For brevity's sake we defer formal security analysis to the full version of the paper. Here we give only a brief, high-level overview.

Completeness is by inspection: a correct execution trace satisfies all of the correctness conditions and therefore satisfies all of the proof sub-protocols.

Knowledge soundness (§2.1, footnote 2) follows from the extractability of the Hyrax polynomial-commitment scheme and the knowledge soundness of data-parallel Spartan and the shift- and grand-product arguments. In broad strokes, the extractor invokes the extractors for each sub-protocol to obtain execution, Block_b, and memory traces. For reasons discussed above ("Protocol details"), the extracted traces satisfy the correctness conditions.

Optimizations. We have described each component of the above proof protocols separately, but many of these proofs can be combined. As examples, we combine fingerprinting for register consistency and the grand product for execution trace compatibility in the same Spartan proof, and likewise collapse the Block_b checks with respect to $\mathcal{C}_b$ for all blocks into a single Spartan proof. In addition, our $\mathcal{P}$ and $\mathcal{V}$ use well-known techniques to batch polynomial commitment opening proofs, taking advantage of the Hyrax commitment's linear homomorphism [72].

4. Evaluation

Our evaluation compares CoBBL's performance to that of a state-of-the-art direct translator and CPU emulator on (a) compile time, (b) prover time (c) verifier time, and (d) proof size. Our baselines for comparison are CirC [62], a direct translator that has inspired several follow-up works [7, 24, 46, 47, 55], and Jolt [9], a CPU emulator developed by engineers supported by Andreesen-Horowitz Research [74, 76]. We discuss our choice of baselines and their configurations below (§4.2).

Our evaluation demonstrates CoBBL's advantages over existing SNARK systems: it outperforms direct translators on programs with complex control flow (e.g., programs with many loop iterations) while dramatically improving compilation time, and it matches or (sometimes dramatically) exceeds CPU-emulator performance, even for computations that are most naturally executed on a CPU. In particular, CoBBL outperforms CirC by 1–30× on compilation time and 26–350× on proving time; its verifier speed ranges from 0.4–8× CirC's. Compared to Jolt, CoBBL's prover time is 1.1–1.8× faster on integer operations (i.e., Jolt's native instructions), and over 100× faster on field operations. CoBBL's verifier time is slower on integer-operation benchmarks, but the differences are within 300ms. Finally, CoBBL produces proofs of smaller sizes than CirC and Jolt on all benchmarks except one.

4.1. Implementation

We build CoBBL's compiler using CirC [62], which provides a framework for building compilers to constraints. CoBBL's front-end comprises an additional 7000 lines of Rust that handles breaking a program into blocks and building the corresponding CFG, optimizing individual blocks using both CirC's existing optimizations and the ones described in Section 3.1, and lowering blocks to constraints using CirC's back-end machinery.

CoBBL's back-end proving machinery is a modified version of Spartan's codebase [3, 65]. Our modifications enable data-parallel proving, which is described in the Spartan paper but was not part of the accompanying code.

Our implementation is open source [5].

4.2. Baselines, benchmarks, experimental setup

This section describes our benchmarks and any changes we made to our baselines to support those benchmarks.

Benchmarks. Figure 1 lists our benchmarks, which we choose both because they've been used to evaluate prior SNARK systems [70, 72, 75] and because they cover complex control flow, field arithmetic, and fixed-width (i.e., 32-bit) arithmetic. We implement each benchmark in two programming languages: we use a Z# [62] version to evaluate CoBBL and CirC, and a Rust version to evaluate Jolt. The two versions of each benchmark are structurally and semantically identical with the exception of field operations, which are natively supported by Z# and implemented using the `ff` crate [78] in Rust. All benchmarks except Poseidon are computed exclusively using 32-bit registers—Jolt's native instruction set—to ensure maximum efficiency for Jolt. We discuss Poseidon further in Section 4.4. For each benchmark and each comparison, we choose parameters (Figure 1) that yield the largest computation for which the baseline system has reasonable running time and memory usage.

Baseline: CirC. We modify CirC [27] to support if-else statements, which our benchmarks require. We rely solely on existing compiler machinery—branching statements are supported in other CirC frontends, e.g., for C)—and modify only the Z# parser and the input format. We configure CirC to use Spartan and the version of the Hyrax polynomial commitment in the Spartan codebase; this essentially matches CoBBL's proof machinery, but CirC does not produce constraints that take advantage of data parallelism.

Baseline: Jolt. We use the Jolt codebase [6] (commit hash `16953f`, from Oct. 22, 2024), compiled in release mode.

Experimental setup. Our testbed is an x86 machine running Ubuntu 24.04 with 12-core Intel i5-1340P chip and 64 GB of memory. For each system and benchmark, we execute the computation five times—recording compile time, prover time, verifier time, and proof size—and average the results.

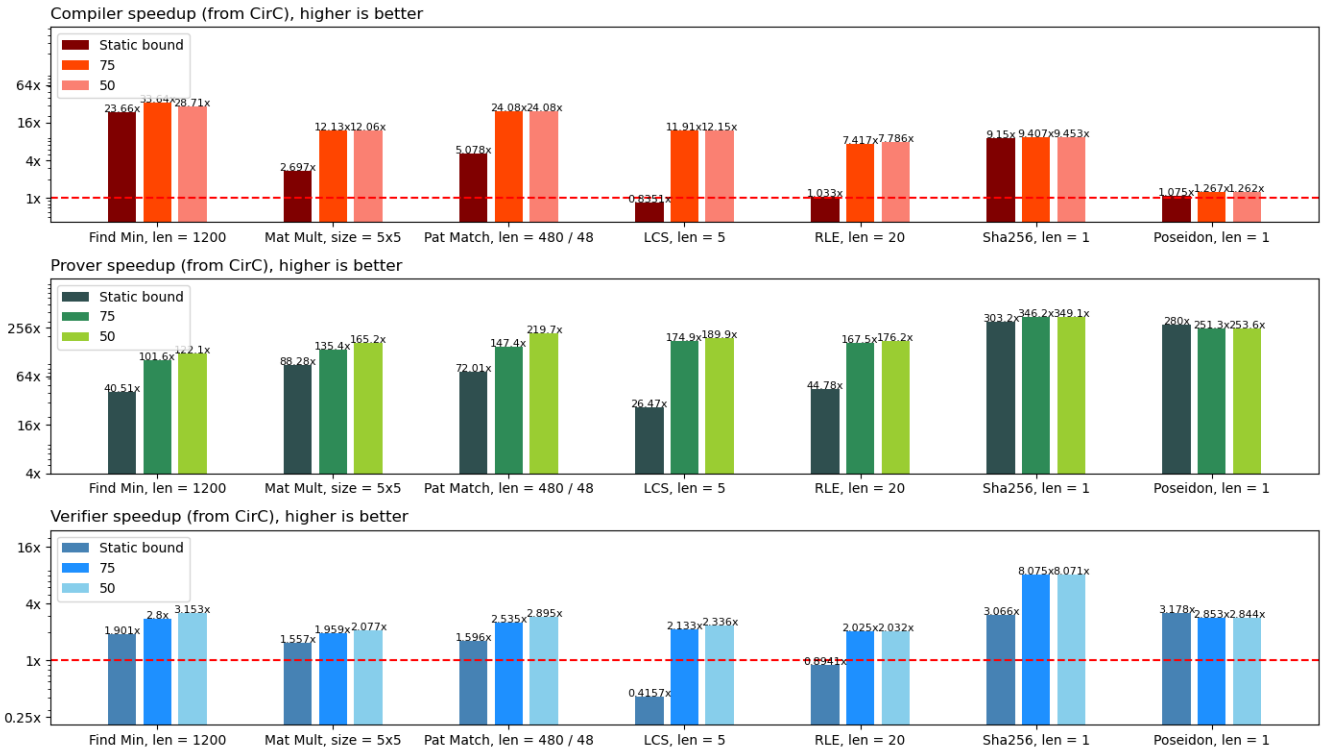

Figure 1: Runtime evaluation of CoBBL, expressed as speedup over CirC (§4.3). Higher is better. CoBBL can avoid unnecessary work depending on program input, but CirC cannot; to demonstrate, we evaluate three scenarios: **Static bound** uses a static bound for both systems; **75** uses an input that requires 75% of the upper bound on loop iterations (which CoBBL can take advantage of, but CirC cannot); **50** uses an input that requires 50% of the upper bound.

Benchmarks	**Parameters**	**Type**
Min value in an array (Find Min)	*v. CirC*: len = 1200 *v. Jolt*: len = 1200	32-bit
Matrix Multiplication (Mat Mult)	*v. CirC*: size = 8x8 *v. Jolt*: size = 16x16	32-bit
KMP pattern match (Pat Match)	*v. CirC*: pat / txt = 48 / 480 *v. Jolt*: pat / txt = 48 / 480	32-bit
Largest common subsequence (LCS)	*v. CirC*: len = 5 *v. Jolt*: len = 30	32-bit
RLE encode + decode (RLE)	*v. CirC*: len = 20 *v. Jolt*: len = 60	32-bit
Sha-256 Hashing (Sha256)	*v. CirC*: len = 1 *v. Jolt*: len = 6	32-bit
Poseidon Hashing (Poseidon)	*v. CirC*: len = 3 *v. Jolt*: len = 6	field

TABLE 1: Overview of benchmarks.

4.3. How does CoBBL compare to CirC?

Figure 1 shows the performance comparison between CoBBL and CirC on a single core. For each benchmark, we measure CoBBL's speedup over CirC on compile, prover, and verifier time across three input scenarios:

1) "*Static bound*": Uses statically bounded array size and number of loop iterations, meaning that CoBBL and CirC prove exactly the same computation. Recall that CoBBL's block-based approach, unlike the direct translation approach, *doesn't* require statically-bounded loops and does not "pay" for unexecuted computation. Therefore, this scenario is the worst case for CoBBL: it is forced to compute on arrays and loops up to the same static bound that CirC uses.
2) "*75*": Executes the dominant loop[7] at 75% of the static bound, and assumes arrays are 75% of their maximum length. This models a scenario where both systems operate on the same inputs, which happen to be 75% of the max size. This means that CirC pays for "unnecessary" computation, i.e., dummy loops.
3) "*50*": Like scenario 2, but replacing 75% with 50%.

Results. CoBBL outperforms CirC in all scenarios across all benchmarks, with the exception of compilation and verifier time for the "static bound" scenario for least common subsequence (LCS) and run-length encoding (RLE). In both cases, CoBBL decides to unroll the most dominant loop, creating substantial compiler overhead and yielding

[7]For matrix multiplication, we only modify the number of rows of the second matrix.

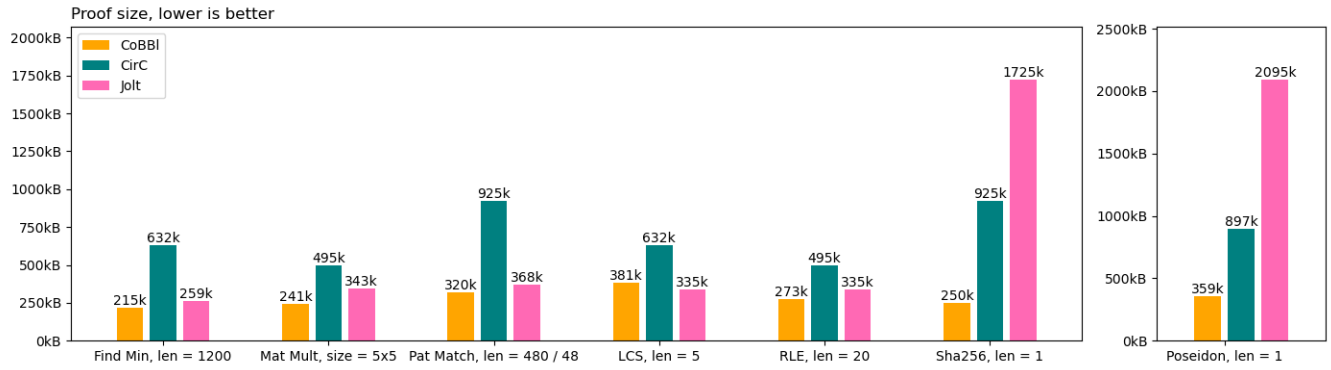

Figure 2: Proof size comparison between CoBBL, CirC, and Jolt. Lower is better.

its advantage on data-parallelism. On the remainder of the benchmarks CoBBL outperforms CirC, achieving 1–30× better compile times and 26–350× better proving times. This is because CoBBL's costs scale only with the constraints actually needed to prove a specific computation, while CirC's costs scale with the maximum number of loop iterations in any valid execution of the program.

Finally, Figure 2 shows a comparison of proof size between CoBBL and CirC (and Jolt, discussed immediately below). It demonstrates that CoBBL emits smaller proofs than CirC across all benchmarks.

4.4. How does CoBBL compare to Jolt?

Figure 3 shows the performance comparison between CoBBL and Jolt. For each benchmark, we record compile, prover, and verifier runtime on a single CPU core.

The results show that CoBBL has *faster* compile times than Jolt, which uses the Rust compiler. Rust's compiler is notoriously slow; we include this comparison not because the absolute times are interesting, but to underscore that CoBBL's compiler performs like a traditional compiler, not like prior direct translators.

CoBBL gives better prover time on all benchmarks that use 32-bit registers (Table 1). This may be surprising given that Jolt is optimized specifically for such computations. Even on these relatively simple benchmarks, Jolt's execution model (i.e., fetching instructions and representing program state with a call stack) appears to impose nontrivial overheads. CoBBL's advantage over Jolt is much more pronounced on Poseidon [43], which comprises mainly field operations. This is because the CPU Jolt emulates does not support field operations, meaning that it must instead emulate field operations (i.e., using a multi-precision arithmetic library); this is extremely expensive.[8]

Jolt's verification speed is usually faster than CoBBL's, by as much as 7.5×. Jolt's choice of polynomial commitment scheme appears to be its main advantage for verification time, and CoBBL may be able to adopt it directly; we discuss in Section 6.

Finally, Figure 2 once again shows that CoBBL produces similar or smaller proofs than Jolt on all benchmarks.

5. Case study: compact certificates

In this section, we examine what CoBBL's performance improvements mean in a practical application. We use *compact certificates* as a case study. Briefly: a compact certificate scheme [60] allows a prover $\mathcal{P}$ to, for example, gather signatures of a large number of attestors on a message, then compress those signatures into a much shorter *certificate* establishing that at least some fraction of signatures are valid. Compact certificates were developed for applications like weighted voting [15, 29, 31, 38]. While compact certificates are already highly efficient, in certain cases it may be desirable to use a general-purpose SNARK to prove knowledge of a valid compact certificate. This would be useful, for example, if the resulting proof were smaller or easier to verify than the underlying compact certificate.

The authors are aware of commercial efforts to build a system for generating "proofs of compact certificate." These efforts have been stymied by exactly the issues discussed in Section 2: CPU emulation is infeasible because of costs, while direct translators were simply not able to compile such a large and complex application. This motivates us to ask if CoBBL makes it possible to compile a high-level description of a very complex computation while avoiding the high overheads associated with CPU emulation. In this section, we answer this question in the affirmative.[9]

Overview. The compact-certificate verification algorithm boils down to performing a sequence of checks, each comprising two Merkle path verifications and one Schnorr signature verification. Compact certificates have the property that, for a fixed number of signers, performing more of these

[8] In the future, Jolt may add support for defining custom extensions; this could be used to accelerate Poseidon at the cost of hand-tuning the extension, then rewriting the program to use it. CoBBL does this automatically with no hand tuning or program rewrite needed.

[9] Recent constructions [30, 35] improve on compact certificates but do not offer drop-in compatibility with existing stake-weighted voting systems. Proof-of-compact-certificate thus remains an interesting application, e.g., as a way of implementing compact blockchain "state proofs" [59].

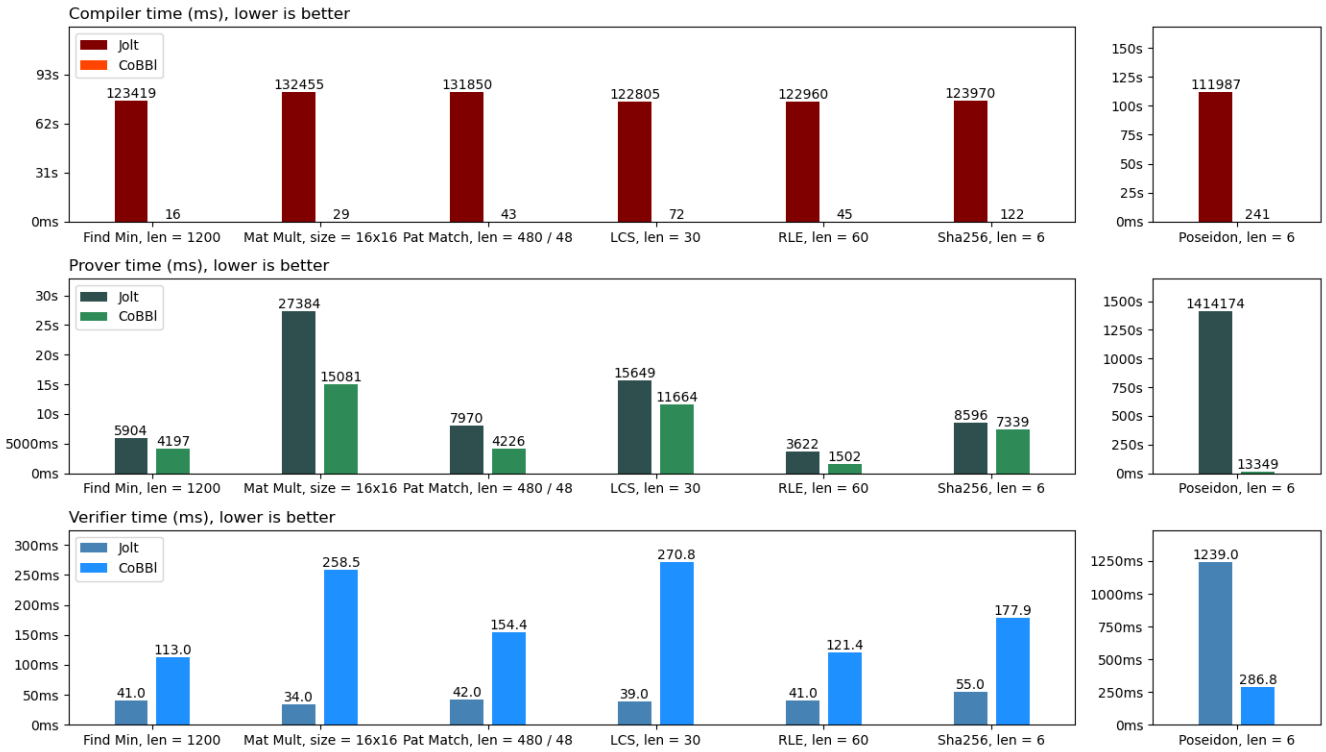

Figure 3: Runtime comparison between CoBBl and Jolt, in milliseconds. Lower is better.

checks convinces the verifier that an increasing fraction of the signers provided a valid signature. Assuming that all signers provide signatures, for example, we have that $\eta = 2^{-\lambda/\chi}$ for security parameter λ, proved fraction of signers η, and χ checks. For example, for security parameter $\lambda = 128$, proving that 90% of signers provided valid signatures requires $\chi = 842$ checks, while a 50% threshold requires 128 checks (independent of the number of signers).

Method. In our case study, $\mathcal{P}$ proves that it knows a compact certificate that the compact certificate verification algorithm would accept. In our experiments, we vary the number of checks χ. We fix the total number of signers as 200,000, which represents a realistic application; performance depends only logarithmically on the number of signers, so the results are within a factor of two even for 2^{30} attestors.

Schnorr signatures in our implementation use an elliptic curve whose base field matches the field over which CoBBl's and CirC's proofs are generated. No matter which elliptic curve we choose, Jolt's computation must do multi-precision arithmetic (§4.4), so the choice of curve doesn't matter; we use the same curve as for CoBBl and CirC.

As in Section 4, we use Z# to implement this computation for CirC and CoBBl, and we use Rust for Jolt. For each system, we select the hash function that gives the best performance: Poseidon for CirC and CoBBl, SHA-256 for Jolt (§4). Finally, because our goal is practicality, we limit proving to 50 minutes and 64 GiB of memory.

Results: scaling. CoBBl is the only system that supports more than a trivial number of checks χ in the compact-certificate verifier computation. Memory, not time, is always the limiting factor. Jolt cannot complete even one check within the time and memory bounds; CirC is able to complete 8; and CoBBl completes 1729 in roughly 2 minutes. CirC outperforms Jolt because it can use native field operations, whereas Jolt must emulate field operations using multi-precision arithmetic. Consistent with the results of Section 4, CoBBl's performance is significantly better than CirC's in a computation with complex control flow.

χ	2	5	8	173	1729
CoBBl	1.49s	1.92s	2.39s	16.7s	134s
CirC	104s	503s	529s	-	-
Jolt	-	-	-	-	-

TABLE 2: Reachable number of checks χ and proving time for the compact-certificates application for CirC, Jolt, and CoBBl (§5).

Results: proof size. To measure CoBBl's effectiveness in compressing the compact certificate, we compare the size of $\mathcal{P}$'s input certificate to the size of the proof of that certificate's validity. We keep the same parameters as in the prior experiment, and we vary the number of checks from 173 to 1729, representing $60\% \leq \eta \leq 95\%$ when $\lambda = 128$. Figure 4 shows the results: when η is low, the certificate is smaller than the SNARK proof. But CoBBl's proofs give better scaling behavior than compact certificates as η grows, with a crossover point between 85% and 90%.

This is not a slam-dunk result, but we are encour-

aged because CoBBl is the first system that is able to prove verification of a nontrivial compact certificate—and it does so *without* requiring experts to hand-craft optimized constraints. Reducing CoBBl's proof size would reduce this crossover point, potentially improving usefulness; we discuss further in Section 6.

6. Discussion, limitations, and future work

CoBBl represents a new design point in the SNARK compilation tradeoff space, combining the optimization opportunities of direct translation, the prove-only-what-you-execute ethos of cutting-edge CPU emulators, and the first program representation and execution model tailored to proving the execution of nontrivial programs. Needless to say, CoBBl is not perfect! In this section, we address (some) limitations and point out (some) future directions.

Expressiveness. As mentioned in Section 3.1, CoBBl's scope stack entails the assumption that every scope entry has a matching scope exit. For many programs this is true; even certain programs that use control-flow constructs that frustrate precise static analyses (e.g., dynamic dispatch) meet this requirement. Nevertheless, a huge class of programs, like JIT-compiled interpreters, appear to be ruled out by CoBBl, but could in principle be used with a CPU emulator. Given the performance characteristics of state-of-the-art CPU-emulator systems, however, proving correct execution of a non-toy JIT looks to be well beyond the horizon. For applications where proof systems are legitimately applicable—where the computation is not so big and is efficiently expressible in the constraint formalism—CoBBl stands to dramatically improve efficiency both during application development and at proving time.

Improving CoBBl's back-end. CoBBl currently builds on Hyrax's polynomial commitment because it gives very fast proving, at the cost of somewhat larger proofs and slower verification than other schemes. This is not fundamental: other multilinear polynomial commitment schemes [49, 56, 61] should be essentially drop-in replacements with different size vs. $\mathcal{P}$ time vs. $\mathcal{V}$ time tradeoffs. In the compact certificates application, for example, 60–80% of CoBBl's proof size is due to polynomial commitments, meaning there is ample opportunity to compress smaller certificates, likely at the cost of additional $\mathcal{P}$ time.

Similarly, alternative RAM abstractions and permutation checks [33, 39, 65] might improve performance, or at least enable different tradeoffs among figures of merit.

Finally, SublonK [26] provides, very roughly speaking, a proof system that avoids paying the cost of untaken branches. It appears that a slightly modified version of SublonK could be used in place of CoBBl's Spartan-based back-end. Extrapolating from the relative performance of Spartan and PlonK-related systems [23, 34], we anticipate this would yield shorter proofs and faster verification at the cost of slower proving.

Incremental and recursive proof systems. Certain systems can prove a recursive or incremental statement, e.g., that a certain property holds on the head of a list and that the prover knows a proof that the recursive statement holds on the list's tail. Recent work on *non-uniform* incremental proofs [51] and on memory abstractions for incremental proof systems [8] point towards another way of achieving, in effect, dynamic constraint generation. CoBBl's general approach to compilation appears to be compatible; future work is exploring incremental proving as an alternative back-end, presumably with different back-end–specific optimizations. We expect that program-specific compilation will offer similar advantages (i.e., program-specific optimizations and state representations) in the recursive proving context.

Zero knowledge. As mentioned in Footnote 1, CoBBl does not attempt to provide a zero-knowledge property. A well-known way to achieve zero knowledge would be to prove knowledge of a CoBBl proof satisfying $\mathcal{V}$'s checks, using a zero-knowledge proof system. This is by no means certain to be concretely efficient, but perhaps CoBBl's proof protocol can be adjusted to be more friendly to this approach.

Hybrid block/CPU execution. Our evaluation shows that CoBBl is competitive with Jolt, a state-of-the-art CPU emulator, even on programs that are most naturally expressed as CPU instructions. There is likely further room for improvement: CoBBl may be able to directly integrate Jolt's approach to executing CPU instructions, thereby giving the compiler a set of efficient general-purpose CPU instructions that can be invoked when the computation calls for it.

Acknowledgments

The authors are grateful to the reviewers, whose insightful comments strengthened the paper. This work was supported by DARPA under Agreements HR00112020022 and HR001125C0300, by Anaxi Labs, and by a CMU CyLab seed grant. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Government or DARPA.

References

[1] a16zcrypto research. https://a16zcrypto.com/research/.
[2] Joltbook: Precompiles. https://jolt.a16zcrypto.com/future/precompiles.html.
[3] Spartan: High-speed zksnarks without trusted setup. https://github.com/microsoft/spartan.
[4] zkSync: trustless scaling and privacy engine for Ethereum. https://github.com/matter-labs/zksync.
[5] https://github.com/cmu-snarks/CoBBl, 2025.
[6] a16z. https://github.com/a16z/jolt, 2023.
[7] Coşku Acay, Joshua Gancher, Rolph Recto, and Andrew C. Myers. Secure synthesis of distributed cryptographic applications. In *CSF*, July 2024.
[8] Arasu Arun and Srinath Setty. Nebula: Efficient read-write memory and switchboard circuits for folding schemes. https://eprint.iacr.org/2024/1605, 2024.

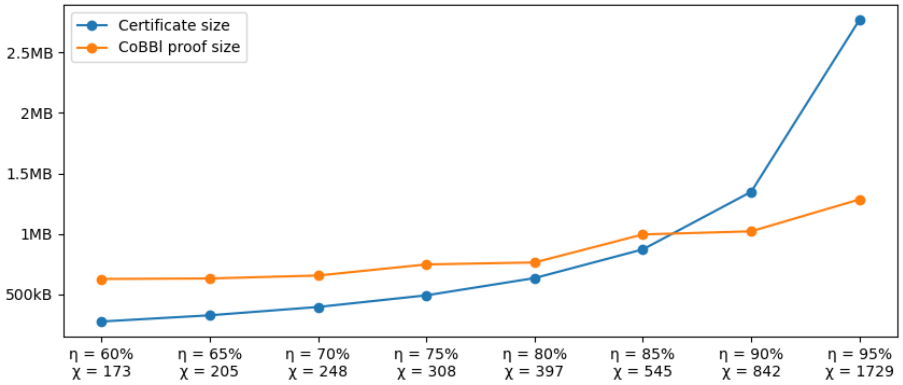

Figure 4: Size comparison between the original compact certificate and the SNARK generated by CoBBl. For each test point, $\mathcal{P}$ uses χ checks to prove that η fraction of signers provided valid signatures.

[9] Arasu Arun, Srinath Setty, and Justin Thaler. Jolt: SNARKs for virtual machines via lookups. In *Eurocrypt*, May 2024.

[10] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptol*, January 2018.

[11] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems: extended abstract. In *ITCS*, June 2013.

[12] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: verifying program executions succinctly and in zero knowledge. In *CRYPTO*, August 2013.

[13] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO*, August 2014.

[14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *USENIX Security*, August 2014.

[15] Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. Cryptocurrencies without proof of work. In *FC*, February 2016.

[16] Brandy Betz. Starkware reaches $8b valuation following latest $100m funding round. https://a16zcrypto.com/posts/article/announcing-a16z-crypto-research/, 2022.

[17] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, January 2012.

[18] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for SNARKs and proof-carrying data. In *STOC*, June 2013.

[19] Manuel Blum and Sampath Kannan. Designing programs that check their work. In *STOC*, February 1989.

[20] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *SOSP*, November 2013.

[21] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent SNARKs from DARK compilers. In *Eurocrypt*, May 2020.

[22] Santiago Campos-Araoz. Introducing RISC Zero. https://risczero.com/blog/announce, 2022.

[23] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. In *Eurocrypt*, April 2023.

[24] Edward Chen, Jinhao Zhu, Alex Ozdemir, Riad S. Wahby, Fraser Brown, and Wenting Zheng. Silph: a framework for scalable and accurate generation of hybrid MPC protocols. In *IEEE S&P*, May 2023.

[25] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: preprocessing zkSNARKs with universal and updatable SRS. In *Eurocrypt*, May 2020.

[26] Arka Rai Choudhuri, Sanjam Garg, Aarushi Goel, Sruthi Sekar, and Rohit Sinha. Sublonk: Sublinear prover plonk. In *PoPETs*, July 2024.

[27] circify. https://github.com/circify/circ, 2020.

[28] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: versatile verifiable computation. In *IEEE S&P*, May 2015.

[29] Phil Daian, Rafael Pass, and Elaine Shi. Snow White: robustly reconfigurable consensus and applications to provably secure proof of stake. In *FC*, February 2019.

[30] Sourav Das, Philippe Camacho, Zhuolun Xiang, Javier Nieto, Benedikt Bünz, and Ling Ren. Threshold signatures from inner product argument: Succinct, weighted, and multi-threshold. In *CCS*, November 2023.

[31] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros Praos: an adaptively-

secure, semi-synchronous proof-of-stake blockchain. In *Eurocrypt*, April 2018.

[32] Edgar G. Daylight. Dijkstra's rallying cry for generalization: The advent of the recursive procedure, late 1950s–early 1960s. *The Computer Journal*, 54(11):1756–1772, November 2011.

[33] Moumita Dutta, Chaya Ganesh, Sikhar Patranabis, Shubh Prakash, and Nitin Singh. Batching-efficient ram using updatable lookup arguments. In *CCS*, October 2024.

[34] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PlonK: Permutations over Lagrange bases for oecumenical noninteractive arguments of knowledge. https://eprint.iacr.org/2019/953.

[35] Sanjam Garg, Abhishek Jain, Pratyay Mukherjee, Rohit Sinha, Mingyuan Wang, and Yinuo Zhang. hints: Threshold signatures with silent setup. In *IEEE S&P*, May 2024.

[36] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Eurocrypt*, May 2013.

[37] Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *STOC*, June 2011.

[38] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: scaling Byzantine Agreements for cryptocurrencies. In *SOSP*, October 2017.

[39] Aarushi Goel, Mathias Hall-Andersen, and Gabriel Kaptchuk. Dora: A simple approach to zero-knowledge for ram programs. In *CCS*, October 2024.

[40] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo, a Turing-complete STARK-friendly CPU architecture. https://eprint.iacr.org/2021/1063.

[41] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. *JACM*, September 2015.

[42] Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S. Wahby. Brakedown: Linear-time and field-agnostic SNARKs for R1CS. In *CRYPTO*, August 2023.

[43] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: a new hash function for zero-knowledge proof systems. In *USENIX Security*, August 2021.

[44] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *Asiacrypt*, December 2010.

[45] Jens Groth. On the size of pairing-based non-interactive arguments. In *Eurocrypt*, May 2016.

[46] Miguel Isabel, Clara Rodriguez-Nunez, and Albert Rubio. Scalable verification of zero-knowledge protocols. In *IEEE S&P*, May 2024.

[47] Kunming Jiang, Devora Chait-Roth, Zachary DeStefano, Michael Walfish, and Thomas Wies. Less is more: refinement proofs for probabilistic proofs. In *IEEE S&P*, May 2023.

[48] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *AsiaCrypt*, December 2010.

[49] Tohru Kohrita and Patrick Towa. Zeromorph: zero-knowledge multilinear-evaluation proofs from homomorphic univariate commitments. *Journal of Cryptology*, October 2024.

[50] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. xJsnark: a framework for efficient verifiable computation. In *IEEE S&P*, May 2018.

[51] Abhiram Kothapalli and Srinath Setty. SuperNova: Proving universal machine executions without universal circuits. https://eprint.iacr.org/2022/1758.

[52] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: recursive zero-knowledge arguments from folding schemes. In *CRYPTO*, August 2022.

[53] Polygon Labs. Polygon Plonky3, the next generation of ZK proving systems, is production ready. https://polygon.technology/blog/polygon-plonky3-the-next-generation-of-zk-proving-systems-is-production-ready, 2024.

[54] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *CGO*, March 2004.

[55] Jan Lauinger, Jens Ernstberger, and Sebastian Steinhorst. zkGen: policy-to-circuit transpiler. In *ICBC*, May 2024.

[56] Jonathan Lee. Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. In *TCC*, November 2021.

[57] Bessie Liu. Chainlight saved zkSync Era from $1.9b exploit. https://blockworks.co/news/exploit-bug-zksync-matter-labs, 2023.

[58] Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, October 2000.

[59] Silvio Micali. State proofs. https://medium.com/algorand/state-proofs-e8c7c2dcb131, 2022.

[60] Silvio Micali, Leonid Reyzin, Georgios Vlachos, Riad S. Wahby, and Nickolai Zeldovich. Compact certificates of collective knowledge. In *IEEE S&P*, May 2021.

[61] microsoft. https://github.com/microsoft/Nova/blob/ad4d77ac89d6bbe9ef943056806e65ceb4ba3b3e/src/provider/hyperkzg.rs, 2024.

[62] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. CirC: compiler infrastructure for proof systems, software verification, and more. In *IEEE S&P*, May 2022.

[63] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: nearly practical verifiable computation. In *IEEE S&P*, May 2013.

[64] Uma Roy. Introducing SP1: a performant, 100% open-source, contributor-friendly zkVM. https://blog.succinct.xyz/introducing-sp1/, 2024.

[65] Srinath Setty. Spartan: efficient and general-purpose zkSNARKs without trusted setup. In *CRYPTO*, August 2020.

[66] Srinath Setty, Benjamin Braun, Victor Vu, Andrew J. Blumberg, Bryan Parno, and Michael Walfish. Resolving the conflict between generality and plausibility in verified computation. In *EuroSys*, April 2013.

[67] Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security*, August 2012.

[68] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In *CRYPTO*, August 2013.

[69] Ioanna Tzialla, Abhiram Kothapalli, Bryan Parno, and Srinath Setty. Transparency dictionaries with succinct proofs of correct operation. In *NDSS*, April 2022.

[70] Riad Wahby, Srinath Setty, Zuocheng Ren, Andrew Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS*, February 2015.

[71] Riad S. Wahby, Ye Ji, Andrew J. Blumberg, Abhi Shelat, Justin Thaler, Michael Walfish, and Thomas Wies. Full accounting for verifiable outsourcing. In *CCS*, October 2017.

[72] Riad S. Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *IEEE S&P*, May 2018.

[73] Dr. Dan Wallach. Securing information for encrypted verification and evaluation (SIEVE). https://www.darpa.mil/program/securing-information-for-encrypted-verification-and-evaluation.

[74] Ali Yahya and Chris Dixon. Announcing a16z crypto research. https://a16zcrypto.com/posts/article/announcing-a16z-crypto-research/, 2022.

[75] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vRAM: faster verifiable RAM with program-independent preprocessing. In *IEEE S&P*, May 2018.

[76] Michael Zhu and Sam Ragsdale. Building Jolt: a fast, easy-to-use zkVM. https://a16zcrypto.com/posts/article/building-jolt/, 2024.

[77] Zeying Zhu, Kan Wu, and Zaoxing Liu. Arya: arbitrary graph pattern mining with decomposition-based sampling. In *NSDI*, April 2023.

[78] zkcrypto. https://github.com/zkcrypto/ff, 2022.

[79] ZKsync. Announcing $200 million in new funding to accelerate the zkSync mission. https://blog.matterlabs.io/announcing-200-million-in-new-funding-to-accelerate-the-zksync-mission-a9d59c1583c8, 2022.

Appendix A. Proof protocol details

A.1. Shift proof

Given committed vectors R and S, this sub-protocol lets a prover $\mathcal{P}$ establish that S is the same vector as R except that it is shifted to the left by a public number of entries with zero padding on the right. That is, if the committed vector $R = a_0, a_1, a_2, a_3$, $\mathcal{P}$ establishes that $S = a_2, a_3, 0, 0$.

To do so, $\mathcal{P}$ first commits to the vectors R and S as univariate polynomials $P_R(z)$ and $P_S(z)$ using any extractable univariate polynomial commitment scheme (§2.2) over a large finite field $\mathbb{F}$. Continuing the prior example, $P_R(z) = a_0 + a_1 z + a_2 z^2 + a_3 z^3$ and $P_S(z) = a_2 + a_3 z$. The proof proceeds as follows:

1) $\mathcal{P}$ sends commitments to $P_R(z)$ and $P_S(z)$ to $\mathcal{V}$, together with the purported values a_0 and a_1, i.e., the constant and linear terms of $P_R(z)$.
2) $\mathcal{V}$ samples and sends a random challenge ζ.
3) $\mathcal{P}$ evaluates $P_R(\zeta)$ and $P_S(\zeta)$ and sends these values along with an evaluation proof for each.
4) $\mathcal{V}$ checks the evaluation proofs and that $P_R(\zeta) = a_0 + a_1\zeta + \zeta^2 P_S(\zeta)$, rejecting if any check fails.

This protocol is complete by inspection. By the binding property of the polynomial commitment scheme, $\mathcal{P}$ cannot equivocate about $P_R(z)$ or $P_S(z)$ after Step 1. The polynomial commitment scheme also guarantees that $P_R(z)$ and $P_S(z)$ have bounded degree d and that $\mathcal{P}$'s claimed evaluations in Step 3 are consistent with the committed polynomials. The polynomial $\Delta(z) = P_R(z) - a_0 - a_1 z - z^2 P_S(z)$ has at most $d+2$ roots in $\mathbb{F}$, so the probability that $\Delta(\zeta) = 0$ (checked by $\mathcal{V}$ in Step 4) when $\Delta(z)$ is not the zero polynomial is at most $(d+2)/|\mathbb{F}|$. This is negligible for large $\mathbb{F}$, meaning that $\Delta(z) = 0$ and thus $P_R(z) = a_0 + a_1 z + z^2 P_S(z)$ with high probability whenever $\mathcal{V}$ accepts. Finally, when the polynomial commitment scheme is extractable, that extractor outputs the vectors R and S with the required relation except with negligible probability, meaning that knowledge soundness (§2.1, footnote 2) holds.

A small optimization. When using the Hyrax polynomial commitment scheme [72], we can make non-black-box use of the polynomial commitment scheme's structure to reduce costs. In this approach, $\mathcal{P}$ commits to a vector T that interleaves R and S, two entries at a time. In other words, $T = R[0], R[1], S[0], S[1], R[2], R[3], S[2], S[3] = a_0, a_1, a_2, a_3, a_2, a_3, 0, 0$.

Using the Hyrax polycommit's inner-product decomposition property, $\mathcal{V}$ can ask $\mathcal{P}$ to evaluate $P_R(\zeta)$ as $\langle T, (1, \zeta, 0, 0, \zeta^2, \zeta^3, 0, 0)\rangle$, for $\langle a, b\rangle$ an inner product. $\mathcal{P}$ can likewise evaluate $P_S(\zeta)$ as $\langle T, (0, 0, 1, \zeta, 0, 0, \zeta^2, \zeta^3)\rangle$. The rest of the proof proceeds as described above. The advantage of this approach is that one length-$4n$ commitment is smaller than two separate length-$2n$ commitments.

Shifts with one-padding. The description above zero-pads S on the right. Our grand-product proof (below) uses a slightly modified shift proof in which S is instead one-padded, i.e., $S = a_2, a_3, 1, 1$ in the above example.

The protocol is nearly identical, with one exception: in Step 4 $\mathcal{V}$ checks that $P_R(\zeta) + \zeta^\ell + \zeta^{\ell+1} = a_0 + a_1\zeta + \zeta^2 P_S(\zeta)$, where $\ell = |R|$. In our protocol $\mathcal{V}$ knows ℓ (§3.2), so this is sound by essentially the same argument as above.

A.2. Grand-product proof

This protocol lets a prover $\mathcal{P}$ establish that $\Gamma_0 = \prod_j \gamma_j$, for a committed set of values γ_j. This works as follows:

First, $\mathcal{P}$ commits to a partial-products vector $T = \Gamma_0, \gamma_0, \Gamma_1, \gamma_1, \ldots, \Gamma_\ell, \gamma_\ell$, where $\Gamma_j = \prod_{i=j}^{\ell} \gamma_i$. That is, Γ_0

is the product of $\gamma_{i\geq 0}$, Γ_1 is the product of $\gamma_{i\geq 1}$, and so on. (Notice that $\Gamma_1\gamma_0 = \Gamma_0$, $\Gamma_2\gamma_1 = \Gamma_1$, etc.) In particular, $\mathcal{P}$ commits to the univariate polynomial $P_T(z)$ induced by the vector T, $P_T(z) = \Gamma_0 + \gamma_0 z + \Gamma_1 z^2 + \gamma_1 z^3 + \ldots$.

Next, $\mathcal{P}$ commits to a polynomial $P_U(z)$ purportedly encoding the vector $U = \Gamma_1, \gamma_1, \ldots, \Gamma_\ell, \gamma_\ell, 1, 1$ that is a left-shift of T by two with one-padding on the right. It then uses a shift proof (described immediately above) to establish that U and U have the required shift relation.

Finally, $\mathcal{P}$ produces a data-parallel Spartan proof over the commitments to T and U proving that $\Gamma_i = \gamma_i \Gamma_{i+1}$ for all $0 \leq i < \ell$. The grand product is $\Gamma_0 = P_T(0)$. We note that Γ_i and γ_i come from the commitment to T while Γ_{i+1} comes from the commitment to U, enabling data parallelism.

By inspection, completeness and knowledge soundness of the grand-product proof follow from the properties of the shift- and data-parallel Spartan proofs.