CoBBl: Dynamic SNARK Constraints using Basic Blocks

Kunming Jiang, Riad Wahby, Fraser Brown

I. Introduction

A SNARK (succinct non-interactive argument of knowledge) is a cryptographical protocol that allows an untrusted prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that it knows a witness z that satisfies certain properties. Trivially, $\mathcal{P}$ can convince $\mathcal{V}$ by sending the entirety of z. Through the usage of SNARK, however, $\mathcal{P}$ can produce a proof with shorter length than z, and $\mathcal{V}$ can verify the proof faster than reading the entirety of z. One popular usage of SNARK is to verify the correct execution of computer programs, which allows users to outsource computations to untrusted parties in cloud computing and blockchain settings.

Early works [SVP+12], [WSH+14], [KPS18], [OBW20] of SNARK primarily focus on program translation. In these *direct-translator* approaches, a program is first converted, in a trusted preprocessing phase, to a set of arithmetic constraints that are satisfiable if and only if the prover $\mathcal{P}$ correctly executes the program. [XXX: Need to emphasize the importance of constraint size.] Next, $\mathcal{P}$ convinces the verifier $\mathcal{V}$ that it holds an assignment that can satisfy the constraints. Since constraint satisfiability is equivalent to correct program execution, $\mathcal{V}$ accepts the output of the program provided by $\mathcal{P}$. Direct-translators have the advantage of utilizing the semantics and structure of a program to produce the constraints most *tailored* to a specific, which can often lead to massive cost reduction. However, the downside is that the constraints produced need to be *fixed at compile time*, and thus have to take into account all execution paths of the program. In practice, proofs generated by direct-translators need to pay for both branches of a conditional statement, and infer and unroll every loop up to a statically-determined upper bound on number of iterations. These proofs often contain *wastes* – work that do not correspond to any executed instructions – increasing compiler, prover, and verifier time.

Later works [ZGK+18], [AST23] explore a new type of SNARK focusing on CPU emulation. Commonly referred to as the *virtual machine (VM)* approach, these systems represent any program execution trace with an instruction set architecture (ISA) like TinyRAM or RISC-V assembly, and express correct program execution through correct execution of individual instructions. They achieve so by pre-generating constraints used to verify each instruction in the ISA, and when later given $\mathcal{P}$'s execution trace, map each instruction in the trace to the corresponding constraints. As such a proof contains only instructions executed by $\mathcal{P}$, it avoids wasted work incurred by direct translators. However, since all constraints are pre-generated, [XXX: How to express the idea that constraints NEED to be fixed at compile time?], SNARK systems employing the VM approach cannot apply program-specific tailoring, conceding a major advantage to their direct-translator counterparts. Furthermore, these systems often require additional wirings in-between the instructions to ensure consistent program states throughout the execution, introducing additional overheads per instruction.

The advantages and drawbacks of the aforementioned two approaches naturally raise a question: *can a SNARK system emit constraints tailored to each specific program like a direct-translator, while paying only for executed instructions like a VM?* We show an affirmative answer by introducing CoBBl, a middle path between the direct-translator and VM approaches that absorbs the advantage of both worlds. CoBBl makes the following innovations and contributions:

1) A SNARK compiler that divides a program into segments and converts each part into constraints.
2) An optimization toolkit that infers the optimal segmentation of a program, and the optimal constraint representation of each segment.
3) A proof protocol that specializes in program segment verification, based on Spartan [Set19].

II. Evaluation

Our experiment with CoBBl aims to answer the following questions:

1) How does CoBBl perform compare to the state-of-the-art direct-translator on (a) compiler time, (b) prover time (c) verifier time, and (d) proof size?
2) How does CoBBl perform compare to the state-of-the-art virtual machine on the four metrics in question 1?
3) How does smaller circuit size produced by CoBBl's block-based abstractions affect performance, in comparison to the overhead introduced by the additional program states?

We choose CirC [OBW20] as the baseline for state-of-the-art direct translator, and Jolt [AST23] as the baseline for virtual machine. [XXX: Need justification? Where to mention that Jolt has an industrial-level budget?] We conduct the experiments on implementations of CoBBl, CirC and Jolt across 7 benchmarks.

A. Implementation

[XXX: Will there be a separate implementation section? Some details might be interesting to go through] We base the frontend compiler of CoBBl on existing infrastructure

of CirC, our direct translator baseline, as CirC contains most underlying functionalities required by CoBBL (in particular, conversion of high-level languages to constraints). On top of CirC, we implement CoBBL's frontend through 7000 lines of Rust code: dividing a program in to segments, performing all optimizations on each segment, and repackaging each segment as individual programs recognizable by CirC's direct translator. We apply minimal modification to CirC's internal codebase to ensure fairness of comparison.

The backend proof system for CoBBL is a custom variant of Spartan [Set19], the same proof system used by our two baselines. We modify Spartan through 7000 lines of Rust code to support parallel execution of all program segments, but leave most internal logic untouched.

B. Baselines and Benchmarks

a) CirC: We modify CirC to support branching statements to align with our benchmarks, using exclusively existing functionalities. Apart from updates to the parser and input format, everything else stays the same as the original codebase [cir].

b) Jolt: Our Jolt evaluation uses the released codebase [a16].

c) Benchmarks: Figure I lists our benchmarks. We implement each benchmark in two programming languages: the Zokrates version is used by CoBBL and CirC, while the Rust version is used by Jolt, compiled with release mode (`--release`). The two versions of each benchmark are identical up to grammatical differences. Since CirC generally performs far worse than Jolt, we choose a different set of parameters when comparing CoBBL against CirC than Jolt. All benchmarks except for Poseidon are computed exclusively using 32-bit registers – the native instruction set for Jolt – to ensure Jolt's maximum efficiency. We explore the special scenario introduced by Poseidon later in the section.

d) Special Benchmark: To answer question 3, we conduct a separate test on the Find Min benchmark, recording detailed performance of CirC and CoBBL for array length ranging from 200 to 1600.

C. Setup

Our testbed is a MacBook pro running on a 10-core M1 Max chip and 64 GB of memory. For each system and benchmark, we execute the computation 5 times, recording compiler, prover, verifier time, and proof size and averaging the results.

D. Method and Results

1) Comparing runtime and proof size of CoBBL, CirC, and Jolt: We present the performance comparison between CoBBL and CirC in figure 1. [XXX: Fix overlap on graph] For each benchmark, we measure the compiler, prover, and verifier time of CoBBL as speedups from CirC across three input scenarios.[1] For CirC to complete more complex benchmarks, we allow polynomial commitment in both CoBBL and CirC to be performed in multicore. Since polynomial commitment takes up a higher percentage of CirC's computation than CoBBL, such a setup yields more advantage towards CirC.

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

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 I: Overview of benchmarks.

The results demonstrate that CoBBL outperforms CirC in all runtime categories across all benchmarks. [XXX: Intend to increase matrix size for Mat Mult.] In particular, CoBBL achieves up to 100x compiler speed up and up to 1000x prover speedup. These results closely match with our expectation: the small constraint size of CoBBL benefits mainly the compiler and the instance commitment phase of the prover.

We demonstrate the performance comparison between CoBBL and Jolt in figure 2. For each benchmark, we record compiler, prover, and verifier runtime on *a single CPU core*. [XXX: Numbers for Poseidon are not updated.] The results show that CoBBL exceeds Jolt by 3 orders of magnitude in compiler time, and for most of the benchmarks operating on 32-bit registers, is on par or better in terms of prover time. [XXX: That compiler time number is embarrassing, should we even mention it?] [XXX: How to discuss the outliers?] For, Poseidon, a benchmark that uses primarily field operations, CoBBL vastly outperforms Jolt on prover time, as field operations are not present in Jolt's instruction set and require emulation using integers. We note that while Jolt's verifier is more efficient than CoBBL , neither incur a runtime over 600ms across any 32-bit benchmarks. Such discrepancy is seldomly taken into consideration in real-life deployment. [XXX: Will there be a chance to address this earlier?] [XXX: Will talk about verifier time once Jolt result for Poseidon is updated.]

We present side-by-side comparison of proof size in figure 3 and demonstrate that CoBBL emits smaller proofs than the two baselines across all benchmarks. [XXX: Again, pending update on Mat Mult and Poseidon.] We further comment that CoBBL is the only system amongst the three that does not

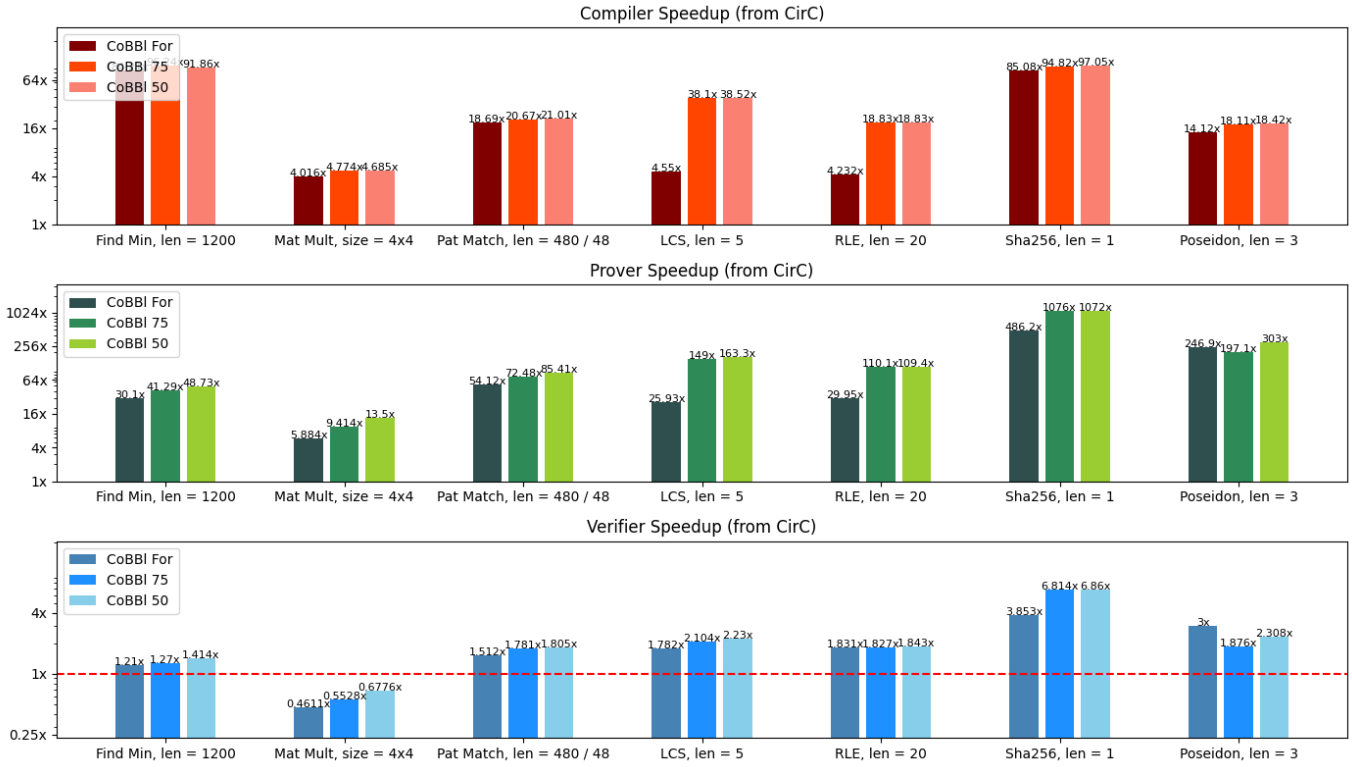

Fig. 1: Runtime evaluation of CoBBL, as speedup from CirC. Since runtime of CoBBL scales with program input but runtime of CirC does not, we derive three scenarios regarding to the workload of CoBBL: **CoBBl For** executes the exact program as CirC, where array size and loop iterations are statically bounded; **CoBBl 75** sets array size and number of iterations on the most dominant loop to be 75% of the upper bound; and **CoBBl 50** sets the same parameters to 50%.

require a static upper bound on number of witnesses.

2) Examining CoBBL*'s improvements in detail:* To answer question 3, we present a detailed analysis of the Find Min benchmark in figure 4. We conduct the experiment by linearly scaling the length of the array from 200 to 1600 while recording runtime and proof size across CoBBL and CirC.

The results demonstrate that while CoBBL incurs a constant factor overhead on number of witnesses and parallel constraints due to the additional program states introduced, it in turn avoids generating and committing to a circuit that expands quadratic to the size of the array. The tradeoff reflects on runtime: CoBBL maintains a constant compile time and linear prover time, asymptotically faster than the quadratic scaling of CirC. For verifier time, the initial constant factor in witness and parallel constraint size slows down the CoBBL verifier, but as array size grows and opening of the circuit commitment dominates the cost of the CirC verifier, CoBBL gradually outcompetes CirC on verifier time. We conclude this special benchmark by remarking that similar patterns are observed across all benchmarks during testing.

References

[a16] a16z. https://github.com/a16z/jolt.

[AST23] Arasu Arun, Srinath Setty, and Justin Thaler. Jolt: Snarks for virtual machines via lookups. Cryptology ePrint Archive, Paper 2023/1217, 2023. https://eprint.iacr.org/2023/1217.

[cir] circify. https://github.com/circify/circ.

[KPS18] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. xjsnark: A framework for efficient verifiable computation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 944–961, 2018.

[OBW20] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. Circ: Compiler infrastructure for proof systems, software verification, and more. Cryptology ePrint Archive, Paper 2020/1586, 2020. https://eprint.iacr.org/2020/1586.

[Set19] Srinath Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. Cryptology ePrint Archive, Paper 2019/550, 2019. https://eprint.iacr.org/2019/550.

[SVP+12] Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Muqeet Ali, Andrew J. Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality (extended version). Cryptology ePrint Archive, Paper 2012/598, 2012. https://eprint.iacr.org/2012/598.

[WSH+14] Riad S. Wahby, Srinath Setty, Max Howald, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient ram and control flow in verifiable outsourced computation. Cryptology ePrint Archive, Paper 2014/674, 2014. https://eprint.iacr.org/2014/674.

[ZGK+18] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vram: Faster verifiable ram with program-independent preprocessing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 908–925, 2018.

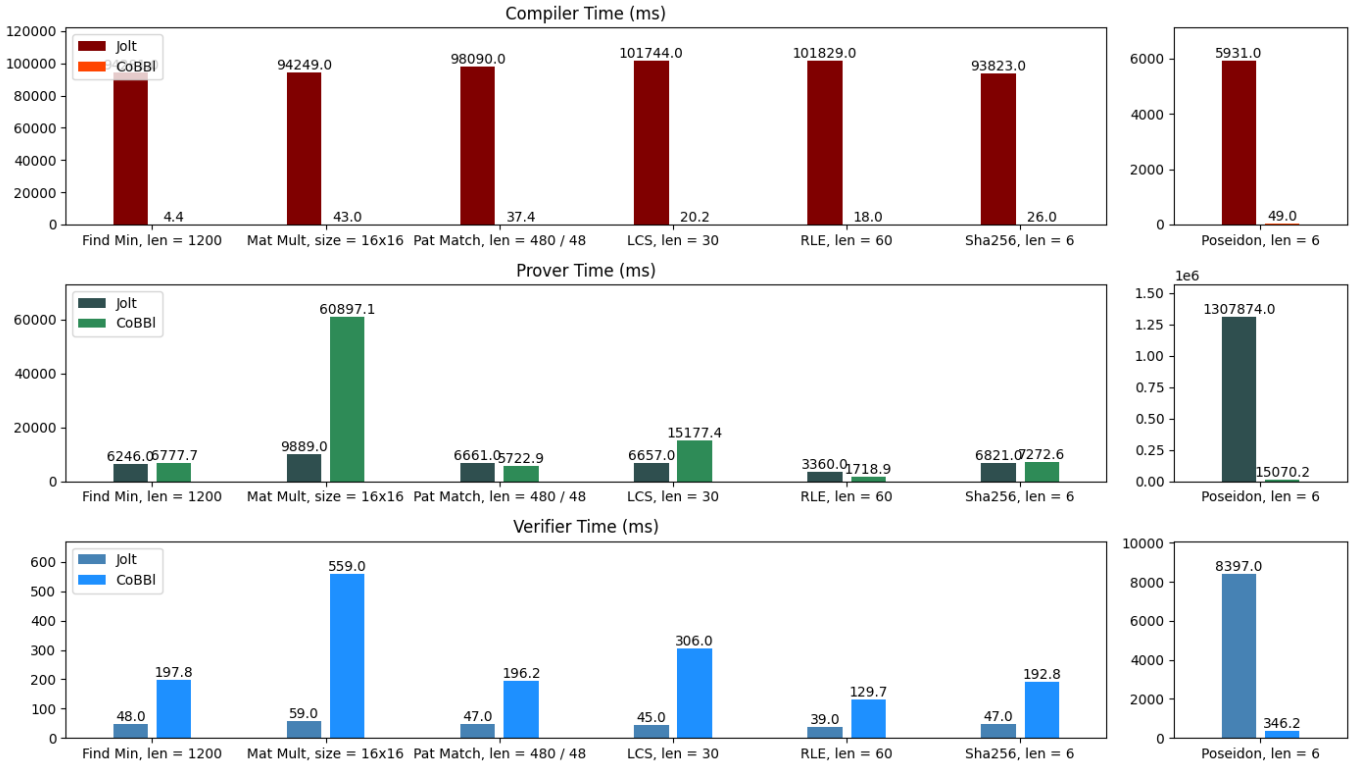

Fig. 2: Runtime comparison between CoBBL and Jolt.

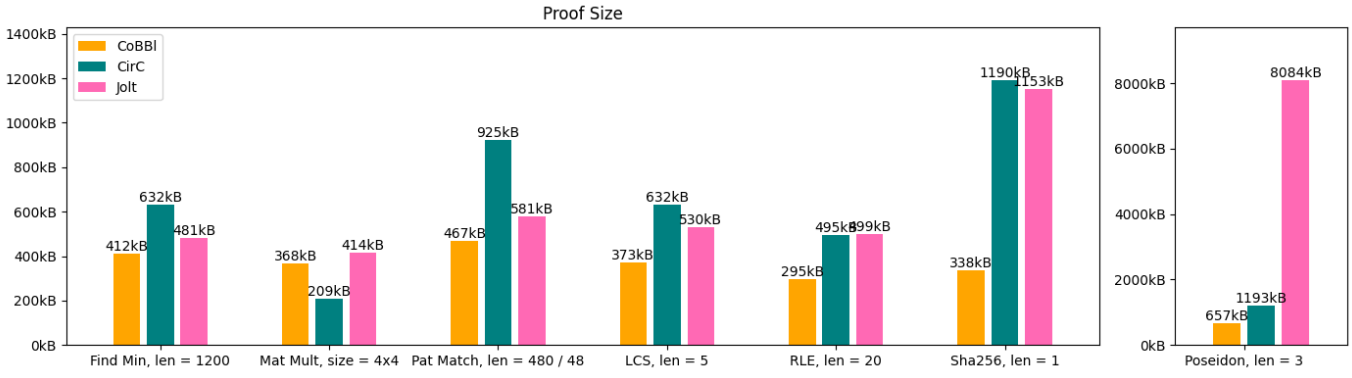

Fig. 3: Proof size comparison between CoBBL, CirC, and Jolt.

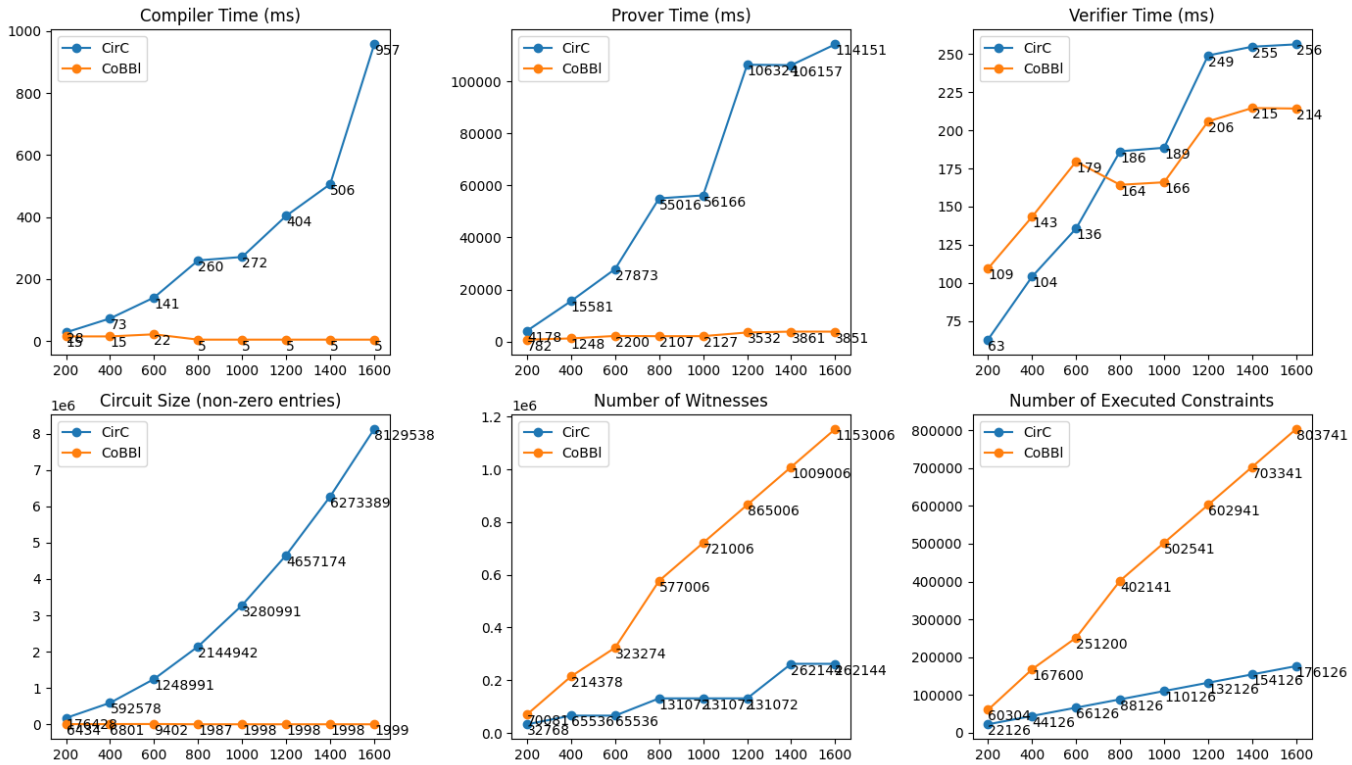

Fig. 4: Time and space costs for CoBBL and CirC on the Find Min benchmark with regarding to array length. **Circuit size** refers to the total size of the circuit emitted by the compiler. **Number of witnesses** marks the total witness size supplied by the prover. **Number of executed constraints** indicates the total amount of parallel constraints checked by the proof.