

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 cryptographic 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 out-source 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 state-of-the-art direct-translators on (a) compiler time, (b) prover time (c) verifier time, and (d) proof size?
- 2) How does CoBBL perform compare to state-of-the-art virtual machines on the same metrics as in question 1?
- 3) To what extent do improvements introduced through CoBBL's block-based abstractions outweigh the overheads?
- 4) How effective are CoBBL's static optimizations in improving proof runtime and size?

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?] We conduct the experiments on implementations of CoBBL, CirC and Jolt across several 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 top of 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, but 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) *COBBL without optimization*: To answer question 4, we disable all non-essential COBBL optimizations (most importantly, block merge and register spilling) and conduct tests across all benchmarks.

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

e) *Special Benchmark*: To answer question 3, we conduct a separate test on the Find Min benchmark, recording 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 of COBBL with CirC*: We present the performance comparison 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. Since

Benchmarks	Parameters	Type
Min value in an array (Find Min)	v. <i>CirC</i> : len = 1200 v. <i>Jolt</i> : len = 1200	32-bit
Matrix Multiplication (Mat Mult)	v. <i>CirC</i> : size = 4x4 v. <i>Jolt</i> : size = 4x4	32-bit
KMP pattern match (Pat Match)	v. <i>CirC</i> : pat / txt = 48 / 480 v. <i>Jolt</i> : pat / txt = 48 / 480	32-bit
Largest common subsequence (LCS)	v. <i>CirC</i> : len = 5 v. <i>Jolt</i> : len = 30	32-bit
RLE encode + decode (RLE)	v. <i>CirC</i> : len = 20 v. <i>Jolt</i> : len = 60	32-bit
Sha-256 Hashing (Sha256)	v. <i>CirC</i> : len = 1 v. <i>Jolt</i> : len = 6	32-bit
Poseidon Hashing (Poseidon)	v. <i>CirC</i> : len = 3 v. <i>Jolt</i> : len = 6	field

TABLE I: Overview of benchmarks.

runtime of COBBL scales with program input but runtime of CirC stays the same, we derive three scenarios regarding to the workload of COBBL:

- **CoBBI For** executes the exact same program as CirC, where the size of the array and number of iterations of each loop is statically bounded.
- **CoBBI 75** sets the array size to be 75% of the statically inferred upper bound. Subsequently, all loops are executed only 75% of the statically inferred number of iterations. [XXX: Need to talk about the exception of Mat Mult.]
- **CoBBI 50** is similar to CoBBI 75, but with only 50% array size.

To enhance CirC’s performance, we allow polynomial commitment in both COBBL and CirC to run in multicore. We note that since polynomial commitment takes up a higher percentage of CirC’s runtime than COBBL, such a setup is more beneficial towards CirC.

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 zk-snarks 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, Muqet 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>.

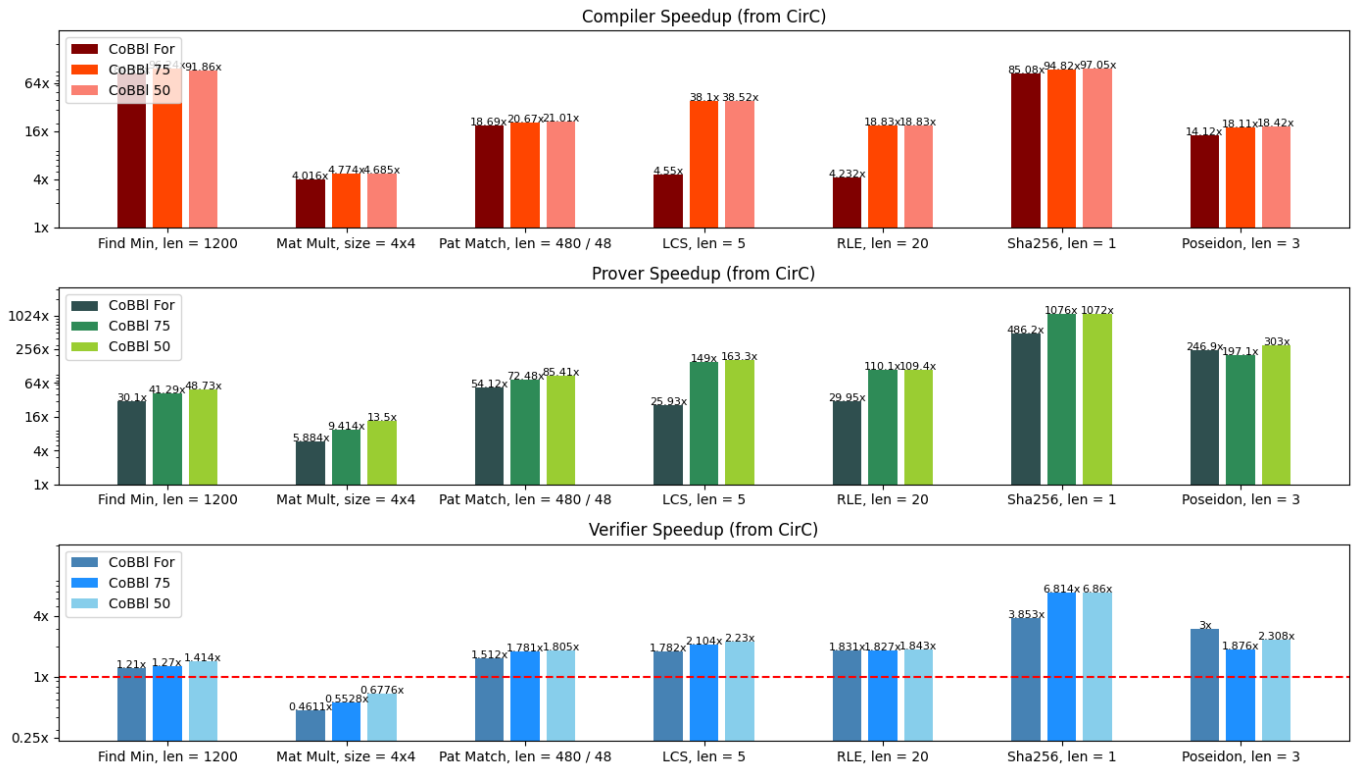


Fig. 1: Runtime comparison between CoBBL and CirC.

- [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.