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:

References

[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.

[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.