CoBBl: Dynamic SNARK Constraints using Basic Blocks

Kunming Jiang

***Abstract*—SNARK (succinct non-interactive argument of knowledge) is a cryptographical protocol widely adopted in cloud computing and blockchain that allows verification of a computation in sublinear time. Existing SNARK systems can be largely divided into two approaches. The compiler approach converts the program into a set of static constraints and generate a proof based on the satisfiability of the constraints, while the virtual machine directly asserts correctness on the sequence of assembly instructions executed by the computation.**

This paper introduces CoBBl, a new SNARK protocol that combines the two existing approaches. Through introducing and modifying the concept of basic blocks, CoBBl achieves all optimizations present in existing SNARK compilers, while allowing the proof to contain dynamic constraints according to the execution path.

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 some property. Trivially, $\mathcal{P}$ can convince $\mathcal{V}$ by sending the entirety of z. However, through the usage of SNARK, $\mathcal{P}$ 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 predominantly focus on program translation. In these *compiler* 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. 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 $\mathcal{P}$. Such compiler approaches are highly optimized, as they can utilize the semantics and structure of the program to produce the most efficient constraints. However, they require the constraints produced to be *fixed at compile time*, which means that they need to take into account all execution paths of the program. In practice, one need convert both branches of a conditional statement into constraints, and unroll every loop up to an inferred or estimated upper bound on number of iterations. Since $\mathcal{P}$ cannot take all branches and loops might break in far fewer iterations than its upper bound, static constraints often lead to "wastes": the proof contains repetitive or unused constraints, increasing compiler, prover, and verifier time.

Recent works [ZGK+18], [AST23] explore a new type of SNARK focusing on CPU abstraction. Commonly referred to as the *virtual machine (VM)* approaches, these systems express any program execution using an instruction set architecture (ISA) like TinyRAM or RISC-V assembly, and opt to verify the correct execution of individual instructions. Proofs of these approaches only include instructions executed by $\mathcal{P}$, thus avoiding the "wastes" introduced by earlier works. However, without the presence of a compiler, SNARK systems employing the VM approach often lacks constraint-level optimization. Furthermore, these systems often impose correctness checks on the program states in between the instructions, further introducing overheads per instruction.

The pros and cons of the two approaches above raise a natural question: *can a SNARK system take advantage of the optimizations available to existing SNARK compilers, while enabling the proof to contain dynamic constraints, like the ones in VM approaches?* To answer this question, we introduce CoBBl, a middle path between the compiler and VM approaches that absorbs the advantage of both worlds. CoBBl makes the following innovations and contributions:

1) A SNARK compiler that converts a program into a set of constraints, each representing a segment of the program.
2) An optimization toolkit that can automatically determine the best segmentation of the program, together with the optimal constraint representation of each segment.
3) A new proof protocol based on Spartan [Set19] that specializes in program segment verification.

II. Preliminaries

To present the general framework in which CoBBl operates, we begin with a few building blocks and general definitions.

a) SNARK: SNARK (succinct non-interactive argument of knowledge) is a cryptographical protocol in between two parties: a prover $\mathcal{P}$ and a verifier $\mathcal{V}$. The input of the protocol is a set of constraints $\mathcal{C}$, to which $\mathcal{P}$ purportedly holds a satisfying assignment z. The protocol itself involves $\mathcal{P}$ utilizing a randomized algorithm to generate a proof of knowledge, which $\mathcal{V}$ can apply the same randomness to verify in time sublinear to the size of $\mathcal{C}$.

b) Verifiable Computation (VC): In a VC setting, a verifier $\mathcal{V}$ outsources a computation $f(x)$ to an untrusted prover $\mathcal{P}$. $\mathcal{P}$ returns the purported output $y \stackrel{?}{\leftarrow} f(x)$ together with a SNARK proof π on the correctness of y. $\mathcal{V}$ checks the proof and either accepts or rejects y. In particular, the above protocol needs to satisfy three requirements:

- **Completeness**: If $y = f(x)$, then $\mathcal{V}$ always accepts y.
- **Soundness**: If $y \neq f(x)$, then $\mathcal{V}$ rejects y except for negligible probability.
- **Succinctness**: The size of π and $\mathcal{V}$'s work should be sublinear to the execution of $f(x)$.

c) Frontend and Backend: SNARK systems are usually divided into two phases: the frontend that generates constraints, and a backend that generates proofs from the constraints. The frontend of compiler-style and VM-style SNARKs are substantially different. However, since these frontends ultimately all emits arithmetic constraints, different SNARK systems can share the same backend.

d) Data-Parallelism: Many SNARK backend systems support data-parallelism, which verifies multiple SNARK proofs at once, with time sublinear to sequential verification. Data-parallelism usually requires the proofs to share some common properties like number of constraints and witnesses.

e) Basic Blocks (BB): A basic block is a sequence of code in a program that contains no control flow in-between. In other words, the execution of the first instruction in a BB implies the execution of every instruction in that BB. BBs are frequently used by compilers as the basis for analyses and optimizations, and a program can be expressed as a control flow graph of its BBs.

III. The Case for a New SNARK Framework

The goal of CoBBl is to explore a middle path between compiler-style and VM-style SNARKs. To achieve so, it takes inspirations from two existing systems from the two approaches: CirC and vRAM. To illustrate the design of CoBBl, we first provide brief descriptions of our two predecessors.

A. CirC

CirC is a compiler infrastructure that provides a link between high-level programming languages and arithmetic or logical circuits like R1CS or SMT. To achieve so, CirC translates a program of a high-level language into a state-free, non-uniform intermediate representation (IR) called CirC-IR. In the context of verifiable computation, these IRs are then lowered into R1CS in a standardized process in-line with those introduced by earlier works [SVP+12], [WSH+14], [KPS18]. Additional optimizations like linear-reductions are deployed to minimize the size of the R1CS constraints.

We build the frontend of CoBBl largely based on CirC. In particular, CoBBl utilizes CirC's translation to and from CirC-IR for its own constraint generation. However, unlike CirC, which translates a flattened version of the entire program into IR, CoBBl invokes additional preprocessing that divides the program into segments, before converting individual segments into IR and R1CS.

B. vRAM

vRAM is a VC system with a circuit-independent preprocessing stage. This means that vRAM does not require any compile-time setup on the program to be verified. Instead, vRAM only requires a one-time setup that generates constraints for every TinyRAM instruction type. For proofs, $\mathcal{P}$ expresses any execution of any program as an *execution trace*: a sequence of program states $[S_0 \dots S_T]$, where the first state S_0 is the program input, and all subsequent states S_{i+1} are generated by applying a TinyRAM instruction I_i to the previous program state S_i. To verify the correctness of the trace, $\mathcal{V}$ checks the following:

- All instructions are fetched correctly.
- All instructions are executed correctly.
- All memory accesses are coherent.

In order to achieve sublinear verification time, vRAM uses data-parallelism to generate a batched proof on all instructions. In concrete terms, it invokes the following procedures:

1) For every state S_i and instruction I_i, $\mathcal{P}$ appends the output of the instruction, O_i, to the end of S_i. The pair (S_i, O_i) form the witnesses for verifying the correct execution of I_i.
2) For every consecutive pair of program states (S_i, O_i, S_{i+1}), $\mathcal{P}$ proves that the corresponding register in S_{i+1} is updated to O_i and everything else stays the same.
3) For instruction correctness, to minimize proof size, $\mathcal{P}$ first permutes the list of (S_i, O_i) by the type of I_i, then verifies that O_i is the correct output of applying I_i on S_i. The permutation allows the proof to only include one copy of constraints for each type of instruction executed. We cover this process in more details in section VI-B
4) Finally, constraints of memory operations extract out all memory accesses. Through a separate permutation, $\mathcal{P}$ proves memory coherency in an approach first introduced by Buffet [WSH+14].

The final proof of vRAM is consisted of four components:

1) Pairwise program state consistency check.
2) Permutation of (S_i, O_i) from execution-order to instruction-order.
3) Batched instruction correctness check.
4) Memory coherence check.

We apply this high-level idea of 4-component proof to the proving structure behind CoBBl, with major changes in each component to match the output of the CoBBl frontend.

C. Motivating for a new SNARK system

While CirC and vRAM are highly efficient in their own regards, they also introduce different overheads in the process. The CirC compiler, as described briefly in the introduction, generates a flattened, static IR for the program. The constraints lowered from such IR need to account for all execution paths, which leads to two problems:

1) The constraints need to express all branches of a conditional, even though only one would be taken in any execution. This creates "wastes" in the proof, *i.e.* constraints that do not correspond to an executed instruction.
2) All loops within the program needs to be flattened and unrolled. Such a process results in constraint size linear to the execution trace, as opposed to size of the code,

inflating the work of the compiler and preprocessor. Further more, loops without an explicit upper bound on number of iterations require hints from the programmer for unrolling; if supplied incorrectly, can compromise proof completeness and soundness.

vRAM solves the above problems by verifying the execution trace and eliminating program-specific preprocessing. In order to do so, however, vRAM introduces program state consistency check. Every instruction I_i in the execution trace translates to one additional (S_i, O_i) entry in the program state permutation and one additional consistency check between S_i and S_{i+1}. Both overheads require linear scan on the program state. For cheap instructions like addition and multiplication, verifying permutation and consistency can be more costly than the instruction itself.

IV. Interpolating CirC and vRAM

We set up CoBBL by first generalizing the achievements of CirC and vRAM. CirC's optimizations are enabled by its static analysis. To retain these optimizations, CoBBL requires a frontend compiler. vRAM avoids constraint waste thanks to its design that permits control flow in-between its constraints – a feature that CoBBL needs to incorporate.

The presence of the compiler necessitates program-specific preprocessing, while inter-constraint control flow implies that the program needs to be segmented in a way that reflects its control flow graph. It turns out there is a natural candidate for the above two requirements: compilers like LLVM [Lat02] perform optimizations by first dividing the code into *basic blocks (BB)*. We note that expressing an execution trace in terms of BBs have the following desirable properties:

1) Each BB contains no control flow in between its instructions. Thus, if the basic block were to be executed, no constraint representing it would be wasted.
2) Typical BBs often span multiple instructions, resulting in reduced number of program states in the trace comparing to the VM approach.
3) There exist extensive researches in the compiler community [SX16], [LHJ+18] on optimizations within and between basic blocks, which can be easily incorporated into the CoBBL compiler.

Despite the stated advantages, basic blocks also come with imperfections. For one, size of BBs of a program can be vastly different. Verifying the instructions of the smallest blocks might still not justify the cost of permutation and program state consistency. Furthermore, data-parallelism in the backend requires all witnesses to be of the same size, leading to extensive padding.

To improve upon BBs, we observe that the trade-off between constraint wastes and program-state check is in fact a continuous spectrum, with CirC and vRAM sitting on either ends of it. While the basic block approach presents itself as a middle path between the two, it is far from being the only alternative. Since conditionals can be translated into constraints, there is no reason why program segments of a SNARK proof need to strictly follow the rules of BBs. Starting from basic blocks, CoBBL performs block merging and register spilling to maximize average proof efficiency. We describe details of these processes in section V. The optimized block are then converted into constraints, which are subsequently fed into a custom-made data-parallel version of the Spartan protocol [Set19].

V. Converting a Program into Constraints

As described in section IV, the CoBBL frontend converts a program into a set of constraints, each corresponding to a segment modified from one or more basic blocks of the program. The entire frontend can be viewed as a three-step process:

1) Convert a program into a control flow graph of BBs and perform basic compiler analyses including liveness, alias, and constant propagation.
2) Perform proof-specific optimizations including block merging and register spilling to generate the final set of program segments.
3) Lower the segments down to a set of IR, then to a set of R1CS constraints.

A. Expressing a program using basic blocks

CoBBL converts a program into basic blocks through a linear scan of its instructions. For every conditional or iteration statement, CoBBL allocates a new block to reflect an edge on the control flow graph. However, unlike LLVM where variables are assigned new versions for each overwrite, CoBBL labels variables by the scope they are defined in. These scope information are later used by register spilling. In addition to the instructions, each basic block also includes a terminator, containing labels of successor blocks and the conditions to reach them.

B. Optimizing blocks for backend

As discussed in section IV, basic blocks still leave rooms for improvement. Below, we identify potential inefficiencies and present CoBBL's solution for further optimization.

a) Block merging: For short basic blocks, the small verification cost for its instructions does not justify the cost to express and check the program state. One simple solution is to merge short BBs together. Block merges reduce the average number of program states at the cost of reintroducing "wastes" back to the proof. To perform block merges, CoBBL first establishes a *block size threshold* based on maximum BB size. Block size threshold helps limit the waste introduced by merges. Next, for every short block, CoBBL scans for merge candidates. If the merged block is still within the size threshold, CoBBL performs the block merge.

b) Register spilling: CoBBL minimizes the number of variables within each program state by excluding out-of-scope variables from the program states. This is achieved through a *scope stack*, where CoBBL selectively pushes out-of-scope variables onto it, and pops them out upon scope changes. In more details, for any variable defined in multiple scopes,

CoBBL initially expresses its value in every scope as separate registers in the program states. Next, CoBBL computes the *maximum program state width*, defined as the maximum number of in-scope variables within any program state. Finally, for every program state that exceeds the maximum width, CoBBL identifies a variable of the outer-most scope (i.e., the variable that is out-of-scope for the longest time), traces to its last and next reference, and pushes and pops the variable on to the scope stack for that duration. CoBBL repeats this spilling process until every program state is within the maximum width.

c) Scope stack: We note that CoBBL's register spilling differs from those employed by traditional compilers, which is usually performed without scoping information. CoBBL's decision to use a scope stack is primarily due to backend efficiency. When translated into constraints, operations on a stack can be expressed much more efficiently than those on a regular memory.

More precisely, CoBBL expresses the scope stack using a write-once memory, in which once a cell is allocated, its value will never be overwritten. Every push corresponds to a new allocation on the write-once memory, while every pop translates to a read. To keep track of the stack frames, CoBBL introduces two new registers: the stack pointer %SP points to the end of the write-once memory, indicating the address for the next allocation (push), while the base pointer %BP points to the beginning of the *previous* stack frame, serving as an offset for the pop operations.

When the program enters a new scope, a new stack frame is initialized, CoBBL pushes %BP onto the stack, and that address becomes the new %BP. Subsequent push operations within the same scope only increment %SP and have no effect on %BP. When the program exits a scope, all values from the previous scope are restored by reading to different offsets of %BP. Finally, %BP itself is restored to the head of the previous stack frame by reading out the value it was pointing to.

We present an example of CoBBL's scope handling in figure 1.

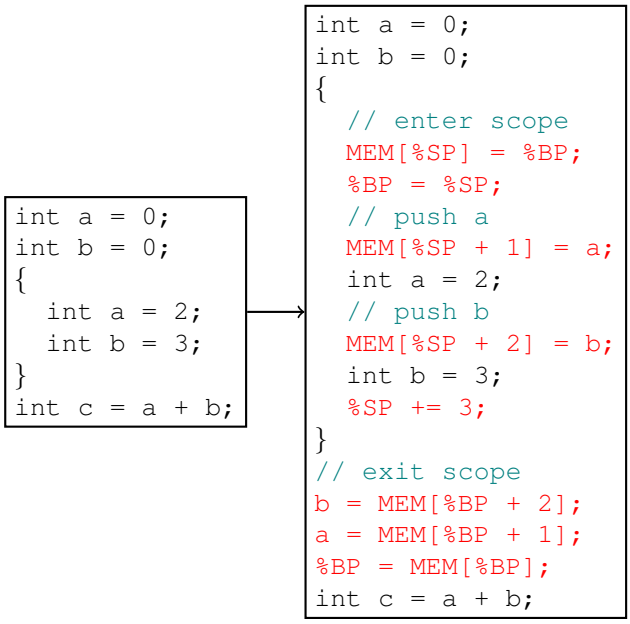

Fig. 1. A scope handling example for CoBBL, where MEM represents the write-once memory used for stack simulation. Note that this transformation takes place only if a and b are selected by CoBBL's register spilling.

C. Lowering segments into constraints

For all non-memory instructions, CoBBL reuses the CirC compiler to lower them down to CirC IR. Memory operations in CoBBL are divided into two categories: those on the scope stack introduced by register spilling, and those on the virtual memory composed of allocated arrays. Virtual memory operations are expressed as in Buffet [WSH+14] in a 4-tuple (addr, val, ls, ts), marking the address, value, whether the operation is a store or load, and the timestamp of the operation. Scope operations are only expressed as (addr, val), as the write-once memory representation ensures that every access to the same address will always yield the same value, regardless of the type of the operation, or when the operation takes place.

VI. Verifying the Constraints Efficiently

The backend structure of CoBBL resembles that of vRAM, and can be divided into four components: correct execution of blocks, consistency of program states, permutation of the program states, and memory coherence. We present the full backend pipeline in figure 2.

A. Execution trace and witnesses

For any execution of the *preprocessed program*, CoBBL expresses the execution trace as a sequence of program states $[S_0 \dots S_T]$, where S_0 is the program input and each subsequent state S_{i+1} is produced by applying a block I_i on the previous program state S_i. However, unlike vRAM, where each instruction sets a single output register, an execution of a block involves arbitrary side effects on all variables and an additional set of intermediate computations. As a result, to verify a block execution I_i, $\mathcal{P}$ computes $Z_i = (S_i, S_{i+1}, w_i)$ as witnesses for the proof, where S_i is the prior program state and the input, S_{i+1} is the next program state and the output, and w_i contains all intermediate computations.

B. Correct execution of blocks

Verification of each individual block execution I_i, when singled out, is the exact same process as verifying a full program: a proof with constraints representing a block and witness represented as $Z_i = (S_i, S_{i+1}, w_i)$. Verification of multiple block executions in a data-paralleled proof, however, involves a few changes. The first is that, just like in vRAM, all executions of the same block need to be grouped together. This introduces a permutation described in section VI-D. Furthermore, number of executions of each block should be a power of 2 (or 0). These two properties allow the binary representation of the ordering to implicitly express the executed block. As an example, assume that the frontend emits 4 blocks: A, B, C, and D. For a particular computation, A is

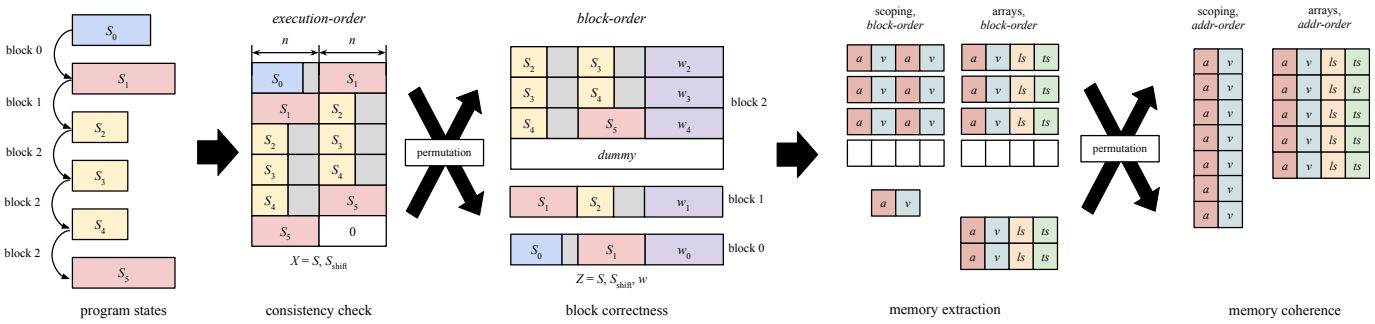

Fig. 2. Overview of the CoBBl backend.

executed 8 times, B is executed 4 times, and C and D 2 times respectively. After sorting by block type, block correctness check verifies 16 executions in parallel, labeled as 0 to 15. To determine which block is applied to each program state, one refers to the binary of the label: if the first bit of the label is 0, then block A is in use; if the first two bits are 10, then B is in use, etc. Such implicit references avoid constraint description within each block execution.

Of course, the number of executions of most blocks are not a power of two. In this case $\mathcal{P}$ pads the proof with dummy executions where every witness is set to 0. This leads to a problem: constraints of every block need to be satisfiable by all-zero witnesses. Fortunately, this requirement is already satisfied by existing SNARK constraints, reasoned below:

- If a constraint does not contain additions that involve constant values (i.e., it either performs arithmetic operations between witnesses, or multiplies a witness by a constant), then it can be satisfied by setting all witnesses to 0.
- To express additions with constant values, SNARK systems typically include a witness assigned to the constant 1 and perform addition using that witness. Since that witness is also set to 0 in a dummy execution, constraints with constant additions are also satisfiable by all-zero witnesses.

Finally, to ensure that $\mathcal{P}$ correctly sorts executions by block type, each block is assigned a label and each program state S_i includes an additional register $\%\mathsf{BN}_i$ indicating the next block to be executed. Each block begins by checking that the value of $\%\mathsf{BN}_i$ matches with its label, and ends by setting $\%\mathsf{BN}_{i+1}$ to the label of the next block. If $\mathcal{P}$ incorrectly sorts or pads the executions, the check on $\%\mathsf{BN}_i$ fails, leading to a rejected proof.

C. Consistency between program states

The consistency check involves verifying that consecutive entries $Z_i = (S_i, s_{i+1}, w_i), Z_{i+1} = (S_{i+1}, s_{i+2}, w_{i+1})$ are consistent with each other, i.e. they share the same S_{i+1}. We note that w_i plays no role in the consistency check, and does not need to be included. We pad each S_i to width $n = \max_i |S_i|$. Let $X_i = (S_i, S_{i+1})$ so $|X_i| = 2n$, the goal of consistency check is to then show that

$$\forall k \in [0, n), X_i[n + k] = X_{i+1}[k]$$

Naively, the consistency proof seems easily parallelizable: each constraint of the consistency proof only uses variables from two consecutive entries X_i and X_{i+1}. One can thus define the witnesses as $Y_i = (X_i, X_{i+1}) = (S_i, S_{i+1}, S_{i+1}, S_{i+2})$ and perform the following check in parallel across all Y_i:

$$\forall k \in [0, n), Y_i[n + k] = Y_i[2n + k]$$

However, such an approach only defers the problem from X to Y. By expressing $Y_i = (X_i, X_{i+1})$, we now have to ask: how to prove that Y_i and Y_{i+1} share the same X_{i+1}?

Instead, CoBBl proves program state consistency through a *proof of shift*. CoBBl represents each X_i as a row in a $2n \times T$ matrix X. As shown in figure 2, the matrix X can be easily divided into two halfs: the left half is a matrix $S = S_0, \ldots S_T$ and the right half is its shifted version $S_{\text{shift}} = S_1, \ldots, S_T, 0$ differed by exactly one row (n entries). CoBBl then generates the proof in the following manner:

1) $\mathcal{P}$ provides S and S_{shift}, and forms $X = S|S_{\text{shift}}$.
2) $\mathcal{P}$ uses rows of X as witnesses to produce a data-paralleled proof on program state consistency.
3) $\mathcal{P}$ proves that S_{shift} is S shifted by n entries.

To prove that S_{shift} is a shift of S, $\mathcal{P}$ interpolates both matrices as univariate polynomials $\tilde{S}, \tilde{S}_{\text{shift}}$ and evaluates them on a random point r. Let $\langle a, b \rangle$ denote vector dot product and

$$R = (1, r, r^2, r^3, \ldots, r^n)$$

then

$$\tilde{S}(r) = \langle R, S_0 \rangle + r^n \cdot \langle R, S_1 \rangle + \cdots + r^{(T-1)\cdot n} \langle R, S_T \rangle$$

$$\tilde{S}_{\text{shift}}(r) = \langle R, S_1 \rangle + r^n \cdot \langle R, S_2 \rangle + \cdots + r^{(T-2)\cdot n} \langle R, S_T \rangle$$

So

$$\tilde{S}(r) = \langle R, S_0 \rangle + r^n \cdot \tilde{S}_{\text{shift}}(r)$$

Since S_0 is the program input already known by $\mathcal{V}$, $\mathcal{V}$ can manually compute $\langle R, S_0 \rangle$ and r^n. $\mathcal{P}$ provides $\tilde{S}(r)$ and

$\tilde{S}_{\text{shift}}(r)$ through polynomial commitment, and $\mathcal{V}$ checks that $\tilde{S}(r) = \langle R, S_0 \rangle + r^n \cdot \tilde{S}_{\text{shift}}(r)$.

Program input and output. To conclude the consistency check, $\mathcal{V}$ needs to additionally check that the program input and output are correctly included in S. This is done by opening all relevant entries in S_0 and S_T and verify their content.

D. Permutation of program states

Since block execution check requires the program states to be sorted by block type, while consistency check requires the program states to be sorted by the order of execution, the next step is to verify that the two transcripts are indeed permutations of each other. Since the intermediate computations w_i are only used in the block correctness proof, w_i does not need to be part of the permutation. The permutation check is then reduced to the following problem: given two list of (S_i, S_{i+1}), how to prove that they are permutations of each other?

CoBBL reuses the idea of Reed-Solomon fingerprinting [Lip90]. Given two lists $a = (a_0, a_1, \ldots, a_T), b = (b_0, b_1, \ldots, b_T)$, to prove that they are permutations of each other, $\mathcal{P}$ constructs polynomials $A(i) = \prod_i (x - a_i)$ and $B(i) = \prod_i (x - b_i)$. $\mathcal{V}$ then evaluates them on a random point τ. If a and b are permutations, then A and B are the same polynomial, so $A(\tau) = B(\tau)$. If a and b are not permutations, then A and B are different polynomials, and the likelihood of them agreeing on a random point is negligibly low.

To expand this idea onto S, where each entry of the lists are program states instead of a single number, CoBBL first performs *the same random linear combination* on each program state to obtain $\text{RLC}(S_i) = \sum_k r^k s_{i,k}$. The remaining steps are now equivalent to that of a regular fingerprinting: construct two polynomials from $\text{RLC}_{\text{block}}$ and RLC_{exec}, and verify the equivalence of their evaluation on a random point τ.

E. Memory Coherence

CoBBL's memory coherence check closely resembles those of Buffet: for every load on an address, the value obtained should be the same as the last value loaded from or stored to that address. To verify this, CoBBL invokes a three-step process: extraction, permutation, and pairwise memory coherence check.

a) Extraction: The first step is to extract all memory accesses from the execution. As described in section V-C, memory accesses in CoBBL are separated into two categories: those on the scope stack and those on arrays. Operations on the scope stack are expressed using the pair $(\mathsf{addr}, \mathsf{val})$, while operations on arrays are described by $(\mathsf{addr}, \mathsf{val}, \mathsf{ls}, \mathsf{ts})$. Fortunately, since constraints for each block is fixed at compile time, for each block execution, CoBBL knows exactly where each memory variable is located. All that remains is for the frontend to append additional constraints to the end of each block, extracting out all scoping and array operations into a list of 2-tuples and a list of 4-tuples.

b) Permutation: The goal of the permutation is to allow $\mathcal{P}$ to sort all memory accesses by their addresses, on which the final coherence check will be performed. We note that this step is inherently identical to the process described in section VI-D. At the end of the permutation proof, $\mathcal{P}$ convinces $\mathcal{V}$ that a list of scoping accesses L_{scope} and a list of array accesses L_{array} contain all memory operations executed during the computation.

c) Pairwise coherence check: With lists of memory accesses purportedly sorted by their addresses, the final check is consisted of two steps. $\mathcal{V}$ first checks that the lists are sorted correctly: for L_{scope}, addresses should be increasing and differ by at most 1; for L_{array}, addresses should be increasing, and accesses on the same address is tie-breaked by the timestamp. The second check is on coherence: for L_{scope}, consecutive accesses on the same address should always result in the same value, and for consecutive accesses on the same address in L_{scope}, if the second access is a load, then the value of the second access should match that of the first. If both checks pass, then $\mathcal{V}$ is convinced of memory coherence.

VII. Conclusion

This paper introduces CoBBL, a SNARK system that combines program-specific optimizations with execution-path-specific constraints. We acknowledge that CoBBL is an enormous system developed by a single grad student, and thus, demands a long development cycle. While a prototype of CoBBL is close to completion, there are still endless opportunities for future optimizations on both the compiler and on creating a more efficient backend protocol. Preliminary testings have shown that the performance of CoBBL is on par with state-of-the-art SNARK compilers. We hope that future work on CoBBL can further improve the efficiency of the system, bringing it closer to industrial standard.

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.

[Lat02] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[LHJ+18] Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. Reconciling high-level optimizations and low-level code in llvm. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.

[Lip90] Richard J. Lipton. Efficient checking of computations. In *Proceedings of the Seventh Annual Symposium on Theoretical Aspects of Computer Science*, STACS 90, page 207–215, Berlin, Heidelberg, 1990. Springer-Verlag.

[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.

[SX16] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 265–266, New York, NY, USA, 2016. Association for Computing Machinery.

[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.