Zero knowledge for computations with RAM *

Dmitry Khovratovich

Sikoba Research

July 4, 2019

1 Introduction

The first zkSNARK proof system with linear prover [PHGR13] handled arithmetic circuits with no memory- or data-dependent operations. All wires were hardcoded to specific variables. As most computer programs have von Neumann architecture with dynamic memory and code addressing, researchers started searching solutions to emulate the von Neumann design with fixed encodings.

This problem is not new, as there have been quite many attempts to emulate such an architecture with finite automata (a Tetris implementation in Conway's Life is a famous example [tet17]). The leading approach is to start with a circuit for a CPU and then prove that it is executed consistently with the memory and code storage.

This idea has been polished in TinyRAM [BCG+13], a simple RISC architecture with flexible word size. To reason about a program, we load it into the memory and consider a full trace of the program execution structured steps with CPU state at each step as a variable. It remains to prove the following:

- For each step its state is computed from the previous one according to the instruction loaded into the CPU for this step and the memory element going into the CPU for this step.
- The instructions are loaded from the program code in the right order. For this, we sort the states by instruction address and prove sequential consistency.
- The memory elements when loaded from the same location are changed only as a result of storing to this location by the CPU. This is also achieved by sorting the states by memory location and proving sequential consistency.

In the next section we provide more details on how exactly this is done. TinyRAM has been implemented in libsnark [lib18].

Alternatives

Buffet [WSR+15] optimizes the [BCG+13] construction by using memory and code subcircuits only for the elements where they are used. VRAM [ZGK+18] is a TinyRAM analog for zkSNARK constructions for small-depth circuits.

*Research supported by Fantom Foundation

2 TinyRAM

In this section we consider a model for dynamic memory and code access used in TinyRAM. The model can be viewed as modelling the internal logic of a RISC CPU with extra logic that verifies the consistency of memory and code retrievals.

2.1 Circuit overview

We construct an arithmetic circuit from an assembly code. As part of the setup, the protocol parties agree on:

- The number r of CPU registers.
- The instruction set $\{I_1, I_2, \ldots, I_s\}$ and the register set S the instructions operate on.
- The number of memory cells is at most T.

For every code C of length l that is compliant with this setting, we construct a circuit $\mathcal{C}$ such that

- If the code outputs O after at most T steps on input I than the circuit also outputs O.

Thus we can reduce the computation integrity problem for C to that of $\mathcal{C}$.

We construct the circuit in the following way.

Circuit variables

1. We allocate l vector variables to describe the program code

$$D_i = \boxed{\text{Timestamp } t \mid \text{Instruction } I \mid \text{Instruction pointer } R_0}, i \in [1; l],$$

   setting $D_i[t] = D_i[R_0] = i$.

2. The program inputs $In[]$ are stored in the first m_0 elements of memory.

3. We allocate a composite variable V_j for step j by attaching the memory bus content M and current instruction I to the register state S_j:

$$V_j = \boxed{t = j \mid S_j \mid I \mid M}, j \in [1, T].$$

We sort the T variables $V_1, \ldots, V_T$ describing the state in two orders:

- Together with $D_1, \ldots, D_l$ by the value of R_0, i.e. we sort by the instruction address. We obtain $T + l$ variables $\widetilde{V_1}, \widetilde{V_2}, \ldots, \widetilde{V_{T+l}}$.
- By the value of R_3 and by timestamp, i.e. we sort by the memory address we work with in the order of appearance. We obtain T variables $\overline{V_1}, \overline{V_2}, \ldots, \overline{V_T}$.

Subcircuits

Suppose we support N instructions in our CPU, including

- JUMP, which set the instruction pointer to the value from register R_1;
- LOAD which set the memory address taken from R_2 to value in R_3.
- STORE which loads to R_3 the content of memory by address taken from R_2.
- Various operations on registers, commonly denoted as COMPUTE$_i$, which implement some function F_i:

$$S \leftarrow F_i(S, M)$$

The circuit consists of:

- T circuits C_{code} :

$$C_{code}(V_j, V_{j+1}) \rightarrow \{true, false\}$$

- T circuits C_{memory} :

$$C_{memory}(\overline{V_j}, \overline{V_{j+1}}) \rightarrow \{true, false\}$$

- $T + l$ circuits $C_{instruction}$:

$$C_{instruction}(\widetilde{V_j}, \widetilde{V_{j+1}}) \rightarrow \{true, false\}$$

- Permutation network C_{perm} that verifies that $\{\widetilde{V_j}\}, \{\overline{V_j}\}$ are permutations of $\{D_j\}, \{V_j\}$ as prescribed above.

2.2 Execution consistency

The C_{code} subcircuit checks that the register set is updated correctly:

- That the instruction register is increased every time by 1 unless we execute JUMP.
- That the memory bus in LOAD is loaded into R_3;
- That the memory bus in STORE is taken from R_3;
- That the COMPUTE instructions execute the correct functions F_i.

The pseudocode of C_{code} is the following:

1. If $C_j == JUMP$ then $S_{j+1}[R_0] = S_j[R_1]$.
2. If $C_j == LOAD$ then

$$S_{j+1}[R_0] = S_j[R_0] + 1;$$
$$S_{j+1}[R_3] = M.$$

.

3. If $C_j == STORE$ then

$$S_j[R_3] = M.$$

.

4. If $C_j == COMPUTE_i$ then

$$S_{j+1} = F_i(S_j, M).$$

.

2.3 Memory consistency

The C_{memory} circuit verifies that LOAD instructions between two STORE calls retrieve the same value and are consistent with STORE.

The pseudocode of C_{memory} is

1. If $C_j \neq STORE$ and $\overline{V_{j-1}}[R_3] \neq \overline{V_j}[R_3]$ (we reuse the memory element) then

$$\overline{V_{j-1}}[M] = \overline{V_j}[M];$$

.

2. If $R_3 < m_0$ and $\overline{V_{j-1}}[R_3] \neq \overline{V_j}[R_3]$ and $C_j \neq STORE$ (we use the input element for the first time) then

$$\overline{V_j}[M] = In[R_3].$$

2.4 Instruction consistency

The $C_{instruction}$ circuit verifies that the same instruction is loaded from the same code location every time.

The pseudocode of $C_{instruction}$ is

1. If $\widetilde{V_j}[R_0] = \widetilde{V_{j+1}}[R_0]$ then

$$\widetilde{V_j}[I] = \widetilde{V_{j+1}}[I];$$

.

2.5 Permutation consistency

The subcircuit C_{perm} consists of two circuit invocations of a Waksman network, with the first implementing the sorting by R_0 and the second one by $R_3||t$. Each subcircuit consists of $T \log T$ switches. We refer the reader to [?] for the details.

2.6 Overall

The outputs of all circuits are combined into a single boolean variable that checks that the result is the same as the alleged O and that all subcircuits return 'true'.

References

[BCG+13] 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 (2)*, volume 8043 of *Lecture Notes in Computer Science*, pages 90–108. Springer, 2013.

[lib18] Scipr lab. libsnark: a c++ library for zksnark proofs., 2018. `https://github.com/scipr-lab/libsnark`.

[PHGR13] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 238–252. IEEE Computer Society, 2013.

[tet17] Quest for tetris., 2017. `https://github.com/QuestForTetris`.

[WSR+15] Riad S Wahby, Srinath TV Setty, Zuocheng Ren, Andrew J Blumberg, and Michael Walfish. Efficient ram and control flow in verifiable outsourced computation. In *NDSS*, 2015.

[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. IEEE, 2018.