TIP 0006: Program Attestation

TIP	0006
authors:	Ferdinand Sauer and Alexander Lemmens
title:	Program Attestation
status:	draft
created:	2022-12-22
issue tracker:	https://github.com/TritonVM/triton-vm/issues/18
pdf:	tip-0006.pdf

Abstract. This note describes an architectural change to Triton VM that (1) ties a proof to the program it was produced from, and (2) gives a program running in Triton VM access to the hash digest of its own program description.

Introduction

Currently, Triton VM is zero-knowledge with respect to the program. That is, the verifier learns nothing about the program of which correct execution is proven. This property is generally desirable, for example to keep a specific machine learning model secret while still being able to prove its correct execution. However, it can be equally desirable to reveal the program, for example to allow code auditing, among other things. This note introduces an architectural change such that

- the output produced when proving a program's correct execution is guaranteed to start with the hash digest of that program's description, and
- the VM's operational stack is initialized with the hash digest of the description for the currently running program.

A program's output is part of the proof for executing said program. Hence, this architectural change ties a program to the proof produced for its execution via the hash digest of said program's description. Any verifier – recursive or not – can choose to compare this digest against an expected digest.

It is worth emphasizing the benefit of writing the same hash digest to two locations, namely standard output and the operational stack: recursive verifiers can compare the hash digest of the proof they are reading to the hash digest *of their own program description*. This way, they can test whether they are actually recursing. Assuming the recursive verifier is sound, verifying a single proof that comes from a chain of thusly generated recursive proofs establishes the correctness of the entire chain up to that point.

Construction Overview

A high-level summary of the construction can be found below. The following sections elaborate on the outlined steps and introduced changes. All steps are part of initializing Triton VM. That is, control is passed to the to-be-executed program only after all the steps described in this note are completed.

1. In the Program Table, hash-input pad the program's description.
2. Copy the program's description from the Program Table into the Program Hash Table.
3. In the Program Hash Table, compute the digest of the program's description by employing Triton VM's canonical hash function in Sponge mode.
4. Copy the computed digest to standard output and into the operational stack's five top-most elements.

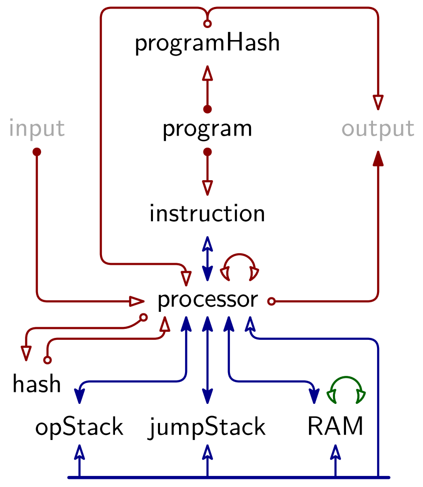

Figure 1: The relation between Triton VM's Algebraic Execution Tables with the proposed construction.

Hash-Input Padding the Program

In order for absorption into the Sponge to work correctly, the program's description needs to be padded such that its length is a multiple of the Sponge's rate `rate`. The employed hash-input padding first appends exactly one 1 and then the fewest possible number of 0s such that the length of the result is divisible by `rate`.

The following base columns are introduced to the Program Table:

1. `absorb_count`, a periodic column counting from 0 to (`rate` $-$ 1),
2. `max_minus_absorb_count_inv`, the inverse-or-zero of (`rate` $-$ 1 $-$ `absorb_count`), and

3. `hash_input_pad_ind`, a bit indicating whether the current element in column `instruction` is part of the hash-input padding.

The already existing column `is_padding` is renamed to `table_pad_ind` to reduce confusion between the two types of padding. The following example illustrates the new mechanics of the Program Table. The used `rate` of 10 is accurate for Triton VM's current hash function.

`address`	`instruction`	`ac`[1]	`mmaci`[2]	`hip_ind`[3]	`table_pad_ind`
0	push	0	9^{-1}	0	0
1	2	1	8^{-1}	0	0
2	call	2	7^{-1}	0	0
3	8	3	6^{-1}	0	0
4	push	4	5^{-1}	0	0
5	10	5	4^{-1}	0	0
6	write_io	6	3^{-1}	0	0
7	halt	7	2^{-1}	0	0
8	push	8	1^{-1}	0	0
9	-1	9	0	0	0
10	add	0	9^{-1}	0	0
11	dup	1	8^{-1}	0	0
12	0	2	7^{-1}	0	0
13	skiz	3	6^{-1}	0	0
14	recurse	4	5^{-1}	0	0
15	return	5	4^{-1}	0	0
16	1	6	3^{-1}	1	0
17	0	7	2^{-1}	1	0
18	0	8	1^{-1}	1	0
19	0	9	0	1	0
20	0	0	9^{-1}	1	1
21	0	1	8^{-1}	1	1
22	0	2	7^{-1}	1	1
⋮	⋮	⋮	⋮	⋮	⋮
31	0	1	8^{-1}	1	1

The following base constraints are added to the Program Table. Note that for reasons of brevity, unchanged constraints are not repeated in this note.

Initial Constraints

1. The `absorb_count` is 0.
2. The `hash_input_pad_ind` is 0.

[1]`absorb_count`
[2]`max_minus_absorb_count_inv`
[3]`hash_input_pad_ind`

Consistency Constraints

1. The `max_minus_absorb_count_inv` is 0 or the `max_minus_absorb_count_inv` is the inverse of ($\texttt{rate} - 1 - \texttt{absorb\_count}$).
2. ($\texttt{rate} - 1 - \texttt{absorb\_count}$) is 0 or the `max_minus_absorb_count_inv` is the inverse of ($\texttt{rate} - 1 - \texttt{absorb\_count}$).
3. The `hash_input_pad_ind` is 0 or 1.

Transition Constraints

1. If the `absorb_count` in the current row is $\texttt{rate}-1$, then the `absorb_count` in the next row is 0. Else, the `absorb_count` increases by 1.
2. If the `hash_input_pad_ind` in the current row is 1, it is 1 in the next row.
3. If the `hash_input_pad_ind` is 0 in the current row and 1 in the next row, then the `instruction` in the next row is 1.
4. If the `hash_input_pad_ind` in the current row is 1, then the `instruction` in the next row is 0.
5. If the `hash_input_pad_ind` in the current row is 1 and the `absorb_count` in the current row is $\texttt{rate} - 1$, then the `table_pad_ind` in the next row is 1.

As polynomials:

1. $(1-\texttt{max\_minus\_absorb\_count\_inv}(\texttt{rate}-1-\texttt{absorb\_count}))\texttt{absorb\_count}'$ $+ \texttt{max\_minus\_absorb\_count\_inv}(\texttt{absorb\_count}' - \texttt{absorb\_count} - 1)$
2. $\texttt{hash\_input\_pad\_ind} \cdot (\texttt{hash\_input\_pad\_ind}' - 1)$
3. $(\texttt{hash\_input\_pad\_ind} - 1) \cdot \texttt{hash\_input\_pad\_ind}' \cdot (\texttt{instruction}' - 1)$
4. $\texttt{hash\_input\_pad\_ind} \cdot \texttt{instruction}'$
5. $\texttt{hash\_input\_pad\_ind}$
   $\cdot (1 - \texttt{max\_minus\_absorb\_count\_inv}(\texttt{rate} - 1 - \texttt{absorb\_count}))$
   $\cdot (\texttt{table\_pad\_ind}' - 1)$

Terminal Constraints

1. The `hash_input_pad_ind` is 1.

Copying the Program

The Program Table's `instructions` are copied to the Program Hash Table in chunks of size `rate`. To prove correct copying, two new Evaluation Arguments and the respective extension columns for them are introduced:

1. the `prep_chunk_eval_arg`, which accumulates one `instruction` per row and is re-initialized every `rate` rows, and
2. the `send_chunk_eval_arg`, which accumulates `prep_chunk_eval_arg` every `rate` rows.

The following extension constraints relative to verifier-supplied challenges α and β are added to the Program Table.

Initial Constraints

1. The `prep_chunk_eval_arg` is $\alpha + \texttt{instruction}$.
2. The `send_chunk_eval_arg` is 1.

Consistency Constraints

None.

Transition Constraints

1. If `absorb_count` in the next row is not 0, then `prep_chunk_eval_arg` accumulates `instruction` in the next row. Else, *i.e.*, if `absorb_count` in the current row is $\texttt{rate} - 1$, `prep_chunk_eval_arg` is set to have accumulated only `instruction` in the next row.
2. The `send_chunk_eval_arg` accumulates `prep_chunk_eval_arg` in the next row if and only if `table_pad_ind` in the next row is 0 and `absorb_count` in the next row is $\texttt{rate} - 1$.

As polynomials:[4]

1. $\texttt{absorb\_count}' \cdot (\texttt{pcea}' - \alpha \cdot \texttt{pcea} - \texttt{instruction}')$
   $+ (1 - \texttt{max\_minus\_absorb\_count\_inv} \cdot (\texttt{rate} - 1 - \texttt{absorb\_count}))$
   $\cdot (\texttt{pcea}' - \alpha - \texttt{instruction}')$
2. $(\texttt{scea}' - \beta \cdot \texttt{scea} - \texttt{pcea}') \cdot (\texttt{table\_pad\_ind}' - 1)$
   $\cdot (1 - \texttt{max\_minus\_absorb\_count\_inv}' \cdot (\texttt{rate} - 1 - \texttt{absorb\_count}'))$
   $+ (\texttt{scea}' - \texttt{scea}) \cdot \texttt{table\_pad\_ind}'$
   $+ (\texttt{scea}' - \texttt{scea}) \cdot (\texttt{rate} - 1 - \texttt{absorb\_count}')$

Terminal Constraints

None.

Cross Table Argument

The terminal value of the `send_chunk_eval_arg` and the corresponding `recv_chunk_eval_arg` in the Program Hash Table need to match. The existing grand cross-table argument is altered accordingly.

Program Hash Table

The Program Hash Table arithmetizes Triton VM's canonical hash function in Sponge mode. This can be achieved by essentially duplicating the existing Hash Table and applying a few yet important changes. Notably, the Program Hash Table computes a single digest for one input of variable length instead of multiple digests for inputs of fixed length. This translates to differences in (a)

[4]For formatting reasons only, `prep_chunk_eval_arg` and `send_chunk_eval_arg` are abbreviated as `pcea` and `scea`, respectively.

input & output behavior, and (b) how the internal state registers update after executing all the hash function's rounds.

The existing Hash Table reads the input for one invocation of the hash function by de-compressing a verifier-randomized linear sum of base field elements that is stored in extension column `RunningEvaluationFromProcessor`. For the Program Hash Table, the to-be-absorbed input is supplied as coefficients of a degree-`rate` polynomial evaluated in the verifier-supplied challenge α (see "Copying the Program") via the extension column `recv_chunk_eval_arg`. This gives rise to a slightly different de-compression function for the Program Hash Table. The details of writing the Program Hash Table's output to the operational stack and to standard output are elaborated on in the section "Copying the Digest".

The existing Hash Table resets its internal state after having written the output, *i.e.*, after having applied all the round functions specified by Triton VM's canonical hash function. Instead of resetting, the Program Hash Table adds the to-be-absorbed elements to its internal state.

Since the existing Hash Table and the Program Hash Table are mechanically very similar, many of their respective constraints are identical. Hence, the constraints regarding round constants, round number transition, and application of the hash function's rounds are not repeated here.

Initial Constraints

1. Register `state_{rate}` is 0.
2. Register `state_{rate+1}` is 0.
3. …
4. Register `state_{rate+capacity-1}` is 0.
5. The `recv_chunk_eval_arg` is $\alpha^{\texttt{rate}} + \sum_{i=0}^{\texttt{rate}-1} \texttt{state\_\{rate-1-i\}} \cdot \alpha^i$.

Consistency Constraints

There are no additional consistency constraints. The Hash Table's constraints resetting some register `state_{i}` do not apply to the Program Hash Table.

Transition Constraints

1. If and only if the round number is 1 in the next row, the `recv_chunk_eval_arg` accumulates $\alpha^{\texttt{rate}} + \sum_{i=0}^{\texttt{rate}-1} (\texttt{state\_\{rate-1-i\}}' - \texttt{state\_\{rate-1-i\}}) \cdot \alpha^i$ relative to challenge β.
2. If the round number is 1 in the next row, all capacity state registers remain unchanged.
3. If the round number is 0 in the next row, all state registers remain unchanged.

Terminal Constraints

See section “Copying the Digest”.

Copying the Digest

With the architectural change suggested in this note, the digest $(d_0, \dots, d_4)$ of the program’s description is embedded in the proof and thus public knowledge. Using the challenge for the Processor Table’s `RunningEvaluationStandardOutput` γ, the verifier can compute $\delta := \gamma^5 + \sum_{i=0}^{4} d_{4-i}\gamma^i$. Hence, proving that the digest was correctly copied from the Program Hash Table to the processor’s operational stack and standard output involves three boundary constraints:

1. Program Hash Table, terminal constraint: $\delta = \gamma^5 + \sum_{i=0}^{4}$ `state_{4-i}` $\cdot \gamma^i$.
2. Processor Table, initial constraint: $\delta = \gamma^5 + \sum_{i=0}^{4}$ `st_{4-i}` $\cdot \gamma^i$.
3. Processor Table, initial constraint: $\delta =$ `RunningEvaluationStandardOutput`.

In the Processor Table, all currently-existing initial constraints involving `st0` through `st4` as well as `RunningEvaluationStandardOutput` are removed.