Audit Report

ZorpZK

Delivered: **May 29, 2023**

Prepared for Zorp Corp. by Runtime Verification, Inc.

Summary

Runtime Verification, Inc. has audited the description of arithmetization of Nock computations for Zorp Corp. The review was conducted from 08/05/23 to 23/05/23.

Zorp Corp. engaged Runtime Verification for reviewing the security and correctness of their approach to arithmetization that lies in the foundation of the new zk-STARK virtual machine for validating Nock computations for the Urbit peer-to-peer network.

Objectives

The main objective of the audit was to analyze the feasibility of the proposed zkVM design and see whether the solution can actually be used for validation of arbitrary Nock computation. Generally, zk-STARK solutions are designed to help a verifier to establish the correctness of a computation performed by a potentially malicious prover, which is done by performing exponentially less work. The Interactive Oracle Proof (IOP) model assumes that the prover's effort is checked within several rounds of sending data commitments (parts of merkle-trees) to a verifier and receiving randomness back. The proposed framework for verifying the integrity of arbitrary programs works as long as the arithmetization (numerical representation) captures all necessary aspects of the underlying computation and the proposed validation technique correctly introduces and utilizes randomness. In other words, the prover will not be able to fake any part of the computation to pass the validation due to the complexity of guessing the right value during validation. Checking that these statements are fulfilled in terms of the proposed design was the primary goal of the audit.

Scope

RV team audited Zorp zkVM design document hosted by Zorp HackMD account. Auditors were provided with other documents used during the audit which served to get a better understanding of the main subject, but were not in the scope of the audit.

Methodology

The audit methodology included manual document review, preparation of diagrams and a formal proof of a bijection between Dyck words and binary trees. The audit process was iterative, and each iteration began with the review of a new section or new content on a daily basis. The RV team provided its feedback at a daily meeting with the client that directed the following changes to the document.

The formal proof of the bijection between full binary trees and Dyck words was prepared in Isabelle/HOL. It shows the correctness of the assumptions made about the algorithms utilized for encoding and decoding full binary trees.

Audit Findings and Conclusion

The suggestions section contains a summary of discussions and provides suggestions for the Zorp zkVM design and its description. The Zorp team addressed posed security and correctness concerns in the final version of the Zorp zkVM design. The proposed Zorp zkVM design meets the principal requirements of a zk-STARK solution and allows proving arbitrary limited-in-size Nock computations. However, the reader should bear in mind that the auditing team worked with the high-level design, and the correctness of the implementation and some parts of the design such as jetting were out of the scope of this audit.

Overall, the proposed zkVM design fully covers key components of the arithmetization of a Nock computation. It introduces the main Nock operations: Nock itself, binary tree concatenation (cons), binary tree decomposition (inverse cons), subtree access (iterated tree decomposition), the identity function, increment, and equality checking. The proposed encoding of full binary trees as Dyck words is accurate, as well as procedures proposed to implement the main Nock operations on the level of binary vectors, including the decoding technique. Arithmetization of nouns as (dyck-word, leaf-vector) pairs and the proposed way of checking the cons relation are robust and valid. Owing to the enormous size of the chosen finite field and the limited size of an input computation, we have come to the conclusion that the use of probabilistic data structures (such as p-stacks and p-multisets) in the zkVM design allows a verifier to succinctly validate whether a computation was computed correctly while rejecting malicious proofs with high probability. The proposed set of tables covers all of the main computational invariants, and comparisons of p-multisets do not leave any gray area for data manipulation. The commitment scheme accurately generates randomness, and situations when the prover can falsify data are avoided with a sufficiently high probability.

Disclaimer

Suggestions and Issues

The main object of the audit is the Zorp zkVM design. The design of the system was stable and was not significantly changed during the audit. However, its description was incomplete at the beginning of the audit. Consequently, technical details, motivation of certain technical decisions, and minor corrections were added later.

Consequently, offered suggestions were related to the specific version of the document reviewed at that time. Significant suggestions and questions were thoroughly discussed and then resolved in the following versions of the design description. Below, we present a summary of the document refinement process. Although auditor feedback and document updates were shared daily, we have separated the description of the audit into three main iterations in the report, each being specific to the state of the document at that time. Each iteration below is supplied with a summary of discussions and provided suggestions.

Iteration I (May 8 - 10)

Document State

The initial version of the Zorp zkVM design contained a high-level overview of the proposed arithmetization of the Nock computation. Overall, sections of the document covered the following matters:

- Full binary trees and their relation to Dyck words. Binary trees are the main data structure that represent data used in Nock computations. Nodes of such trees always have two siblings. Hence, such unbalanced trees are called *full binary trees*. Such trees store values only in their leafs. Trees are called nouns in the context of Nock computations. However, nouns are encoded as Dyck words in practice. Such binary vectors are generated by a simple depth-first tree traversing. There is also a procedure that allows decoding Dyck words back to trees. These relations form a bijection between Dyck words and binary trees. In practice, there are two vectors required to encode a tree: a Dyck word to record the shape of the tree and a leaf to store leaf values.
- Overview of the zk-STARK protocol. The main goal of the zk-STARK protocol is providing means for validation of an arbitrary computation carried out by a potentially malicious prover without the necessity to repeat the whole computation. The protocol relies on an interactive oracle proof (IOP) model. The technical implementation of the protocol requires representing the computation using polynomials over a finite field.
- Description of the finite field used for proposed arithmetization. All nouns used in Nock computations are encoded as field elements using polynomials according to algorithms mentioned below.

- The Schwartz–Zippel lemma. It allows estimating the probability of finding a root of a polynomial with coefficients in the aforementioned field. The property is crucial for estimating security soundness of the proposed solution.
- P-multisets (or nd-multisets in the original version of the document). A multiset is a data structure similar to a set, but it can contain several instances of the same value. A p-multiset is a method of encoding and comparing multisets using randomness generated by a verifier in terms of a zk-STARK protocol. Values of a p-multiset of size *n* are encoded as roots of a polynomial. Such polynomials allow validating equivalence between two multisets with a high probability without actual comparison of elements.
- Tuple compression. Tuple compression is a technique where *k* field values can be compressed into a single value using *k* random values received from the verifier. Tuples are encoded as simple linear multivariate polynomials. For example, values *x, y, z* can be compressed with randomness *a, b, c* as *ax* + *by* + *cz*.
- P-stacks (or nd-stacks in the original version of the document). Such stacks are not actually stacks, but rather a technique of computing a field element for *n* values using a single random value received from a verifier. The main procedure of the technique is placing values of interest in the coefficients of a polynomial of the following form:

  $$s(\alpha) = s_0\alpha^n + \cdots + s_{n-1}\alpha + s_n.$$

  Where α is the randomness received from the verifier. The technique has the name of a stack because it can be intuitively considered as a stack where values are pushed or popped only from the top because Horner's method allows efficiently computing or verifying the result of adding or removing a single element.

- Description of noun representation. Nouns are a key data structure in Nock computations. A noun can be represented using a Dyck word and leaf vector. Dyck vectors can be represented recursively, and the *cons* relation describes how to make a new Dyck word *W* using two Dyck words *L* and *R* relating them as follows: $W = 0\,L\,1\,R$. Arithmetization of the cons relation using polynomials is based on properties of polynomials used for p-stacks. Consequently, the cons relation can also be checked at the level of field elements using specific constraints described in the document being reviewed.
- Description and an example of a Nock computation. The section briefly and informally describes the Nock computation using a few logical operators over nouns.
- High-level description of tables used for arithmetization of a Nock computation. There are noun, memory or Nock 0, stack, decoder, exponentiation tables. The pop table was added later during the audit.
- The noun table is the main table used for validation of nouns. Base columns contain nouns used in a computation in the form of Dyck word and leaf vector literals, including leaf vector lengths. Extension columns contain computed Dyck and leaf vector p-stacks and compressed triples of the length of the leaf vector and p-stacks. The final computed

column contains p-multisets for compressed tuples. These p-multisets are used for comparing values from different tables for validation of used nouns.

- Notably, the original version of the document used a 3-stage commitment scheme for generating randomness that was changed two times later. The original version assumed generation of only a single random value, used to compute p-stacks; the second commitment is followed by generation of random values for tuple compression and computation of p-multisets.

Review

The RV team started the document review by covering the basics of the Nock data representation and reviewing definitions of binary trees and Dyck words. An informal proof of a bijection between Dyck words and full binary trees was constructed. The following step was checking the use of the Schwartz–Zippel lemma and understanding the relationship between zk-STARK arithmetization and similar solutions and protocols that were used as inspiration. At the next step, the probabilistic data structures were reviewed, and this step was followed by understanding the noun table, and especially reviewing the correctness of using probabilistic data structures in its design.

RV team provided the following suggestions to refine the content of the document during the first audit iteration:

- Asked to add more details about missing tables and illustrate their work with examples. The initial version of the document was also missing a description of both base and extension columns of all tables except the noun and memory tables.
- Proposed to use the same example of a tree and corresponding noun in all sections.
- Asked to explicitly show that Dyck words used in the document do not correspond to Dyck words matching parentheses expressions to avoid confusion.
- Proposed a rigorous definition of a binary tree, similar to their definition of a Dyck word.
- Proposed to slightly improve the statement of the Schwartz–Zippel lemma, which is now used in the latest version of the document.
- Suggested to elaborate more on the motivation behind the choice of the finite field.
- Noted that the document does not provide any information about jetting with respect to arithmetization.

Iteration II (May 11-16)

Document State

The main contribution in the second version of the document was the description of the memory table. This table allows verifying correctness of memory accesses which reduces to checking the cons relation between subtrees. Memory access can be thought of as traversal down a specific path in a binary tree from its root to the required subtree or leaf. Such a path is represented by a specific binary vector called an *axis*. The value 0 means choosing the left subtree, and 1 is the choice of the opposite direction. The memory table verifies that the cons relation holds for all nouns computed after processing each digit of the axis.

Another table described during this iteration was the exponentiation table. This table is responsible for precomputing powers of randomness used to validate the cons relation in various tables.

The document was light on detail when describing the base columns of the memory table, but provided enough details and illustrations to comprehend computation of p-stacks and p-multisets. The latter serves for cross validating values computed by different tables.

Review

The RV team focused on two major objects: the use of nondeterministic data structures, and the design of the noun, memory, and exponentiation tables. The first item led us to review the choice of the finite field and the necessity of limiting the potential size of any computation explicitly. We also discussed the necessity of using p-multisets for data validation between tables. The review of the tables' design was followed by the discussion of the number of commitments and dependencies between columns. We discussed possible attack vectors that could be exploited by a malicious prover. The last step of the iteration was reviewing the design of the exponentiation table. Finally, we suggested the following changes:

- Suggested to bound the computation size explicitly, as the Schwartz–Zippel lemma can be safely used with polynomials of degree less than the size of the field. The size of the chosen field is huge in practice, but theoretically, it is possible to create a recursive computation that would end up with enormous nouns.
- Proposed to add cons relation description for leaf p-stacks.
- Suggested to formalize the Nock description, agreed that it is just enough to add the reference to the existing complete definition.
- Requested to explicitly cover how an incorrect computation is handled by the memory table.
- Asked to add more explicit arguments about the choice of the commitment scheme.
- Proposed to explicitly mention that the constraint for the lengths in the provided example of the memory table is redundant, but tracking lengths is very helpful for optimizing computations.
- Proposed to explicitly mention that there is a separate p-multiset comparison between the memory and stack table, as there are more such comparisons between other tables.
- Proposed to use the same example of a tree and corresponding noun in the table examples as ones used in other sections.
- Noted that the exercise related to permutations of a noun table is theoretical, and it is not a part of the protocol.

- Noted that the fourth cell in each row of the memory table containing the compressed tuple of p-stacks and leaf vector length is computed later than the other three elements with the current commitment scheme.
- Proposed to explicitly say that constraints in the memory table are written regarding the length of the leaf vector, as it would explain differences from formulas from the p-stack section.
- Proposed to explicitly add how the memory table uses multiplicity for "caching" repreated computations.
- Suggested to explicitly describe how the exponentiation table uses multiplicity, and highlight that it is done for optimization, not for security.
- Proposed to add how the memory table gets the appropriate nouns matching specific subtrees without necessarily putting nouns explicitly in the memory table base columns.
- Proposed to explicitly show the timeline of commitments and column computation stages.

Iteration III (May 17-19, 22-23)

Document State

Previous stages covered mostly tables related to data used during the computation. Sequentially processing each opcode of the program is the responsibility of the decorder and stack tables. Their description was provided during the last iteration of the audit.

The stack table implements a stack machine with the stack height bounded by 2^{32} to handle Nock opcodes recursively. The state of the computation is defined by the computation stack with a subject and formula to process, a product stack containing the results of the computation and binary flags that define the current mode.

The decoder table helps to destructure formulas for the stack table, which guides the direction of the computation. Invalid formulas and unsupported opcodes are caught during the computation of the decoder table.

The pop table was also added in this iteration. Its purpose is to validate the correctness of popping operations performed in the stack table.

Another valuable addition to the document was a new diagram that illustrates multiset comparisons between each table. The diagram tracks all dependencies between tables. Another diagram of data dependencies between columns and corresponding randomness illustrated the use of randomness in one place.

An important change that was made in the latest version was changing the number of commitments. The latest version has two commitments with a slightly modified randomness scheme compared to the original version. The main difference is the generation of randomness for tuple compression after the first commitment to the base columns of all tables.

Review

The RV team was focusing on the review of new table descriptions and changes to the number of commitments. We thoroughly discussed the stack machine model and reviewed the popping mechanism and subformula computations. Results of the review led to requiring the introduction of a new table known as the pop table. Another focus of the iteration was the error processing that happens in decoder and memory tables. The newly written descriptions of these new tables urged us to return to the relation of these tables with the noun table.

During the iteration, we reviewed and discussed the number of commitment stages and new arguments in favor of reducing the number of commitments. The final commitment scheme has two commitments and randomness generation that overall breaks down into three computation phases that, omitting some technical details, correspond to the preparation of base columns, computation of p-stacks and tuples, and then computation of p-multisets. The main change in the commitment scheme is the generation of randomness for tuple compression after the first commitment. This decision doesn't reduce the security of the protocol since all values are known to the verifier: original nouns are committed to, p-stacks and tuples are computed with values provided by the verifier.

At this stage, the RV team was working on a formal proof of a bijection between Dyck words and binary trees. The direction of the proof was changed after discussion of equivalence between Dyck words in general and Dyck words constructed using a recursive pattern.

The RV team provided the following suggestions:

- Asked to add more details about the popping mechanism of the stack table.
- Proposed to add more explicit descriptions for commitment stages and affected columns, since the related description is scattered across several sections.
- Proposed to add explicit description of usage of multiplicity in the pop table.
- Suggested to describe each multiset comparison per an arrow in the diagram illustrating table interrelations.
- Asked to elaborate more on new random values mentioned in the new version.
- Suggested to explicitly explain which metadata is collected during the Nock execution.
- Proposed to explain error processing that may happen during a computation and its reflection in the ZK computation.
- Suggested to clearly define the comparison of p-multisets between tables.
- Suggested to add a high-level introduction of how Nock computation and ZK validation are connected.
- Asked to describe base columns for each table.

- Proposed to explicitly mention multiplicity, as it was omitted in some places and mentioned in others.
- Asked to add more technical details of how noun validation between the noun table and the pop table is done.
- Proposed to reorder sections, but the actual plan was provided later.

Bijection Between Trees and Dyck Words

The RV team prepared a formal proof in Isabelle/HOL that establishes the fact that the depth first search algorithm and the decoding algorithm are inverse bijective maps between arbitrary Dyck words and full binary trees.

We refer to an arbitrary binary vector V that meets the following two properties as Dyck words:

1. $\forall\, k \le length(bv)\colon\ occurrences(1,\ prefixOfLen(k,\ V)) \le occurrences(0,\ prefixOfLen(k,\ V))$
2. $occurrences(1,\ V) = occurrences(0,\ V)$

The first property means that the number of occurrences of 1 in any prefix of the vector is less or equal to the number of occurrences of 0 in this prefix. The second one states that the numbers of occurrences of digits 1 and 0 in the vector are equal.

The main idea of the proof is preparing an inductive definition of so-called ZDyck words used extensively in Nock computations for encoding full binary trees:

$zdyck = [] \mid 0 \# zdyck\ @\ 1 \# zdyck$

The first part of the proof shows that the ZDyck words definition is equivalent to the Dyck word definition, that enables the use of inductive proofs. The equivalence is also important for the definition of the *cons* relation from the theoretical perspective. The next step is proving that depth first traversal always returns a Dyck word. The full binary tree encoding procedure actually guarantees that the output binary vector meets the ZDyck definition. Then we show that the relation between the depth first traversal dfs and the proposed algorithm for decoding full binary trees $totree$ are mutually inverse:

$\forall\, \mathrm{t} : totree(dfs(t)) = t$

$\forall\, \mathrm{bv} : dyck(bv) \rightarrow dfs(totree(bv)) = bv$

The final part of the proof shows that both functions are injective and surjective. That leads us to a bijection in turn:

$\forall\, \mathrm{x\ y} : dfs(x) = dfs(y) \rightarrow x = y$

$\forall\, \mathrm{y} : \mathrm{dyck(y)} \rightarrow \exists\, \mathrm{x} : dfs(x) = y$

$\forall\, \mathrm{x\ y} :\ dyck(x) \wedge dyck(y) \rightarrow (totree(x) = totree(y) \rightarrow x = y)$

$\forall\, \mathrm{y} : \exists\, \mathrm{x} : \mathrm{dyck(x)} \wedge totree(x) = y$