ZERO KNOWLEDGE MEMORY-CHECKING TECHNIQUES FOR STACKS AND QUEUES

ALEXANDER (SASHA) FROLOV

Abstract. There are a variety of techniques for implementing read/write memory inside of zero-knowledge proofs and validating consistency of memory accesses. These techniques are generally implemented with the goal of implementing a RAM or ROM. In this paper, we present memory techniques for more specialized data structures: queues and stacks. We first demonstrate a technique for implementing queues in arithmetic circuits that requires 3 multiplication gates and 1 advice value per read and 2 multiplication gates per write. This is based on using Horner's Rule to evaluate 2 polynomials at random points and check that the values read from the queue are equal to the values written to the queue as vectors. Next, we present a stack scheme based on an optimized version of the RAM scheme of Yang and Heath [16] that requires 5 multiplication gates and 4 advice values per read and 2 multiplication gates per write. This optimizes the RAM scheme by observing that reads and writes to a stack are already "paired" which avoids the need for inserting dummy operations for each access as in a stack. We also introduce a different notion of "multiplexing" or "operation privacy" that is better suited to the use case of stacks and queues. All of the techniques we provide are based on evaluating polynomials at random points and using randomly evaluated polynomials as universal hash functions to check set/vector equality.

1. Introduction

Running an arbitrary RAM program in a Zero-Knowledge Proof is a long-studied problem. Papers like Pantry [4], or BCGT 2013 [2] are early works in this area, using Merkle Trees or permutation checks respectively to check consistency of memory access results provided as advice values by the prover. Later work on implementing memory using permutation checking has substantially optimized circuit sizes and comparisons, eventually achieving constant overhead for RAM accesses [7]. [16] has optimized this to 6 multiplication operations and 4 comparisons per access[1].

There is also some work implementing related data structures, such as sets [10], multisets [5], maps [15], and Read-Only Memory [12]. In general, the fastest techniques for memory in Zero-Knowledge Proofs use evaluations of polynomials at random points. These techniques (and the ones presented in this paper) use randomness and interaction together to achieve their results.

There is also some limited work on implementing data structures related to the ones discussed in this note, like stacks and linked lists. The Reef paper [1] and [13] present two techniques for implementing a stack in zero-knowledge as part of executing a type of finite automata used in the paper. We will sketch the two techniques presented by the Reef paper in Section 4. ZkPi [12] presents a system for implementing a data structure called a "linked list", though accesses require a constant number of circuit gates and it does not support deletion.

In this note, we present techniques for memory checking for stacks and queues based on similar polynomial techniques to previous papers. Our queue scheme most closely resembles the memory techniques of ZkPi, while the stack technique can be thought of as an optimized version of the memory techniques of [16]. They are more efficient in terms of circuit size than prior work, and

[1]This is in the limit where the number of memory accesses is much larger than the size of the address space (so the setup/teardown phases are amortized), and where multiplication gates with one public input don't count towards the total (these are used in a step where random linear combinations of tuples of field elements are computed).

our scheme for queues completely avoids permutation checking, which is somewhat novel among polynomial-based memory checking techniques.

2. Background

2.1. Universal Hash Functions.

2.1. Universal Hash Functions. We use the terminology of ZkPi [12] for the *coefficient hash* and *root hash*.

Define the root hash of a vector of elements $\overrightarrow{x} \in \mathbb{F}^n$ to be $H_r(k, \overrightarrow{x}) = \Pi_{i=0}^{n-1}(k - x_i)$. When evaluated at a random point independent of the choice of $\overrightarrow{x}$, this is a universal hash function with collision probability $\frac{n}{|\mathbb{F}|}$ (for inputs that differ as multisets). Additionally, it can be incrementally evaluated by multiplying a running hash value by $(k - x_i)$ if all elements of $\overrightarrow{x}$ are not known at the beginning of a computation.

Likewise, define the coefficient hash of a vector of elements $\overrightarrow{x} \in \mathbb{F}^n$ to be $H_c(k, \overrightarrow{x}) = \sum_{i=0}^{n-1} k^i x_i$. This is also a universal hash function with collision probability $\frac{n-1}{|\mathbb{F}|}$ for inputs that differ as vectors. This can also be evaluated incrementally by Horner's Rule as $H_c(k, \overrightarrow{x} + x_n) = k * H_c(k, \overrightarrow{x}) + x_n$ (where $\overrightarrow{x} + x_n$ denotes appending x_n to the vector).

2.2. Model for Memory-Checking Arguments. As in previous techniques like Spice's memory checking [15], we specify our techniques as interactions between an untrusted prover storing a data structure and a verifier performing some procedure to verify that the prover is faithfully storing the data structure.

Such a procedure can be converted to a procedure for zero-knowledge proofs where the circuit run by the prover implements the technique to verify the results of storage operations provided as advice in the witness. This allows for a zero-knowledge proof circuit to soundly access a data structure by performing a low-cost procedure to validate that inputs in the witness represent valid storage operations.

2.3. Notion of Operation Privacy for Queues and Stacks. If the pattern of accesses to a queue or stack is known ahead of time, then "trivial" stack/queue schemes that require no multiplication gates or advice values are possible. In this case, the location where a value is read from the stack/queue can be connected to the location where it is written with a wire. These locations are known in advance because the pattern of accesses to the stack/queue is known.

Consequently, the pattern of accesses to a stack or queue must be data-dependent for the problem of designing a stack/queue data structure for zero-knowledge proofs to be nontrivial. This property is called "multiplexing" or "operation privacy" in prior work. In previous papers on RAM in zero-knowledge proofs, the primitive used is a "multiplexed" `access` operation, which takes an operation type (read or write), address, and value (to be used if performing a write) as arguments. Depending on the operation type argument, this `access` will perform a read or a write on the RAM.

One can naively port this idea to a stack or a queue, but the resulting primitive is difficult to reason about. A directly analogous `access` method would either add or remove an element from the data structure depending on its operation type. Both read and write operations to a stack/queue mutate the data structure. Such an `access` method would be inconvenient to program with, since such an interface does not give the option to not change the data structure.

Instead, we propose "conditional pop" and "conditional push" operations (and analogous operations for queues). The Reef system uses its stack data structure in this way [1], as does [13]. A conditional pop operation takes a "guard" argument, and performs the operation if guard is true, and otherwise does nothing. Likewise, a "conditional push" operation takes a guard argument and the value to be pushed, and pushes it to the data structure if the guard is true, doing nothing otherwise. This still makes the patterns of accesses to the stack/queue data-dependent, while being easier to reason about and program with. It is trivially possible to build a "conditional write" primitive from the commonly-used interface for RAM mentioned above, and this seems a way that these primitives are commonly used.

2.4. **Cost Accounting for our Schemes and Related Work.**

Our scheme for stacks builds on some components in [16]. We will refer to this paper and its memory scheme as the Yang-Heath RAM scheme.

We account for the costs of our scheme in the same way as the Yang-Heath scheme(s). The Yang-Heath scheme's cost accounting focuses on *non-linear gates* and *advice values* as the costs for a memory scheme. This is because most SNARK systems' costs are dominated by the costs of non-linear operations.

The Yang-Heath scheme requires taking random linear combinations of tuples of values as part of checking that 2 vectors of tuples of values are permutations of each other. The Yang-Heath scheme considers the random linear combinations to be linear operations, since the random coefficients for the linear combinations are public. We will inherit this method of accounting, but count the number of random linear combinations, since these may still count as multiplications in some proof systems.

3. Memory-Checking for Queues

A queue is a "First-In First-Out" data structure. The ith value written to a queue must be the ith value read from a queue. Thus, queues have a much simpler notion of memory consistency than a RAM or other data structures. There is not a notion of address, and memory consistency only requires checking 2 conditions: 1) that the vector of values enqueued to the queue is the same as the vector of values dequeued from the queue and 2) that the queue is not empty during every pop operation. We will first specify and analyze a protocol that does not keep operation types private in Figure 1 and then explain how to make the scheme private to hide operation types.

```
type Queue_$\ell$ {
  $enqueues : \mathbb{F}$
  $dequeues : \mathbb{F}$
  $queue\text{-}depth : \mathbb{N}$
  $depth\text{-}check : \mathbb{F}$
  $r : \mathbb{F}$
  $\alpha : \mathbb{F}^\ell$
}
setup-queue() $\to$ Queue_$\ell$ {
  // $r, \alpha$ must be sampled independently of advice values
  $r \xleftarrow{\$} \mathbb{F}$
  $\alpha \xleftarrow{\$} \mathbb{F}^\ell$
  $m \leftarrow$ Queue_$\ell$ $\{0, 0, 0, 1, r, \alpha\}$
  return $m$
}
  // Performs operation when $g$=1, no-op otherwise
enqueue($q$ : Queue_$\ell$, $v : \mathbb{F}_\ell, g : \{0,1\}$) {
  $q.enqueues \mathrel{+}= g * (q.enqueues * r + \langle v, \alpha \rangle - q.enqueues)$
  $q.queue\text{-}depth \mathrel{+}= g$
}

  // Performs operation when $g$=1, no-op otherwise
dequeue($q$ : Queue_$\ell$, $g : \{0,1\}$) $\to \mathbb{F}_\ell$ {
  $v \leftarrow$ input_$\ell$( )
  $q.depth\text{-}check \mathrel{*}= (q.queue\text{-}depth + (1 - g))$
  $q.dequeues \mathrel{+}= g * (q.dequeues * r + \langle v, \alpha \rangle - q.dequeues)$
  $q.queue\text{-}depth \mathrel{-}= g$
  return $v$
}

teardown($q$ : Queue_$\ell$) {
  for $i \in [q.queue\text{-}depth]$ {
    $v \leftarrow$ input_$\ell$( )
    $q.dequeues \leftarrow q.dequeues * r + \langle v, \alpha \rangle$
  }
  assert $q.depth\text{-}check \neq 0$
  assert $q.queue\text{-}depth = 0$
  assert $q.enqueues = q.dequeues$
}
```

Figure 1. Our ZK queue data structure. Uses of non-linear gates are highlighted. Note that r, α are sampled independently of the prover's advice values (or after they are committed to).

3.1. **Soundness analysis.** Now, to analyze the soundness of the scheme, let T the number of conditional enqueue operations performed on the queue. Let us analyze this for $\ell = 1$ first. The `enqueue` and `dequeue` methods are both evaluating degree T polynomials *enqueues*, *dequeues* at r with Horner's rule (note that r is independent of the values enqueued/dequeued by construction). *enqueues* is obviously degree T, and `teardown` will make the degree of *dequeues* T. Call the T values that are enqueued $e_1, ..., e_T$ and the dequeued values $d_1, ..., d_T$. By Horner's rule, *enqueues* will be the polynomial $e_T + e_{T-1} * x + e_{T-2} * x^2 + ... + e_1 * x^{T-1}$ evaluated at point r and *dequeues* will be the polynomial $d_T + d_{T-1} * x + d_{T-2} * x^2 + ... + d_1 x^{T-1}$ evaluated at point r. If the prover performed consistent accesses to the queue, the ith value dequeued will be equal to the ith value enqueued, so these two polynomials will be equal with probability 1. If the prover performed inconsistent accesses and dequeued a value at step i that wasn't the ith value added, the two coefficient vectors e, d will disagree. By applying the Schwartz-Zippel lemma, *enqueues* and *dequeues* will be equal with probability $(T-1)/|\mathbb{F}|$ if they have different coefficients. If $\ell > 1$, then *enqueues*, *dequeues* are the evaluation of $T - 1 + \ell$-degree multivariate polynomials (we can view the ℓ elements of α as the values of ℓ separate variables of degree 1). Again, α is independent of the prover's advice values, so we can apply the Schwartz-Zippel Lemma. By applying the multivariate Schwartz-Zippel Lemma, the soundness error for $\ell > 1$ is at most $(T - 1 + \ell)/|\mathbb{F}|$.

Secondly, by inspection, we can see that *q.queue-depth* tracks the depth of the queue at every point. For any execution that reads from the queue when it is empty, *q.queue-depth* must be 0 and g must be 1 during some read (since the depth only changes by 1 at each read and the guard must be 1 for a read to happen). This means that *q.depth-check* will be 0 iff the queue was read from when it was empty, because it will have been multiplied by 0 during a call to `enqueue`. Thus, the checks performed by the verifier guarantee that the queue always has at least 1 element in it when being read from, and guarantee with negligible soundness error that the order of reads and writes to the queue are consistent.

3.2. **Cost Accounting.** Without hiding operation types, this scheme uses 1 multiplication and 0 advice values per call to `enqueue`, and 2 multiplication and 1 advice value per call to `dequeue`. Both operations also use $O(\ell)$ linear operations for computing additions and $\langle v, \alpha \rangle$.

Including the costs for conditional operations, there is an additional cost of $O(1)$ linear operations per enqueue/dequeue and 1 multiplication per enqueue/dequeue to multiply by g, giving the stated costs in the abstract.

Note that this scheme allows for operations with private operation types and public types to be mixed.

The setup phase uses a constant number of operations, while the teardown phase does $O(n)$ work, where n is the maximum possible depth the queue reaches. Concretely, per remaining entry in the queue, the teardown phase takes 1 advice value, performs one multiplication, performs, some conditional operations, and performs $O(\ell)$ linear operations. To be completely general, a circuit for the teardown phase would have to handle n remaining entries.

However, most queue-based algorithms that we are aware of completely empty the queue at the end of execution, so we believe it is reasonable to assume that the teardown phase's gate requirements are negligible relative to the main program's execution, and can be assumed to be negligible in computing costs per operation[2].

4. Memory-Checking for Stacks from existing primitives

As baselines to compare against, we describe some simple techniques for implementing a stack in a zero-knowledge proof.

[2]The Yang-Heath RAM construction makes a similar assumption about the costs of the setup/teardown phases being amortized.

4.1. Reef's Stack Memory Techniques.

We first describe the existing stack schemes presented in the Reef paper (these are also presented in [13]). Reef's programming model is slightly different from the one presented in this paper, as Reef uses the Nova folding scheme [11]. The library used does not support commit-and-prove techniques, and evaluating permutation-check polynomials for all memory accesses is infeasible because the folding scheme splits the computation across multiple folded instances.

The first scheme presented, which we call `HashStack`, is based on a cryptographic hash function (Reef uses Poseidon [9]) and essentially performs hash chaining. The verifier maintains a digest D representing the stack's state. To push a value v, the verifier computes the updated digest $D' = H(v, D)$. To pop a value v, the prover provides digest D' as advice, the verifier checks that it satisfies $D = H(v, D')$, and the digest is updated to D'.

This is a constant-overhead scheme. It requires 1 advice value per pop. However, it requires a Poseidon evaluation for each push/pop, which requires a few hundred constraints to implement [9].

The second scheme that they present, which we call `WireStack`, can be thought of as an application of the "linear pass"-style RAM which was used in some early SNARK papers [6]. In the model of a verifier outsourcing storage to a prover, this can be thought of as a trivial scheme where the verifier just stores the stack themselves. In a circuit, this involves maintaining n wires (where n is the maximum depth achieved by the stack), and performing linear passes to find the first non-empty element per push/pop. The authors of Reef claim that the wire-based approach is faster than the hash-based approach for small stacks.

4.2. RAM-based memory checking for stacks.

A simple memory-checking technique on top of an existing RAM technique is possible, requiring 1 additional multiplication per pop and only additional linear operations per write in addition to the per-access costs of a ZK RAM scheme. This can be done using the "depth-check" technique of the previous section. Given a contiguous region of dedicated RAM in an existing RAM scheme, this scheme can maintain accumulators *stack-depth* and *depth-check*. When performing a conditional push (with $guard = 1$ representing doing an operation and $guard = 0$ representing not doing anything), the verifier must update *stack-depth* as $stack\text{-}depth + guard$, and write the value pushed to address *stack-depth*. Likewise, when performing a conditional pop, the verifier must update *depth-check* to $depth\text{-}check * (stack\text{-}depth + (1 - guard))$, then update *stack-depth* as $stack\text{-}depth - guard$ and perform a read at address *stack-depth* and return the value. During the teardown phase, it should be asserted that *depth-check* is nonzero.

Per pop, this scheme requires 1 additional multiplication in addition to the underlying RAM scheme's costs. This scheme uses 7 multiplications and 4 advice values per read when instantiated using the Yang-Heath RAM scheme. Values are read/written in the correct order because of the soundness of the underlying RAM scheme, and the stack depth is always nonzero because the depth check approach would flag reads from an empty stack as justified before. The depth check approach lets us check that a series of values is never zero with 1 multiplication per check.

5. Memory Checking for Stacks

Finally, we present a memory-checking technique for stacks based on an optimized version of the Yang-Heath memory checking technique presented in [16], though some of the ideas presented originate in [3] and [15].

Let T be the total number of (conditional) push/pop operations performed on the stack, and let n be the maximum depth the stack ever reaches during a computation (for consistency with the Yang-Heath paper). We make a few observations about stacks that allow us to optimize the costs of this scheme:

- In a valid sequence of accesses to a stack, a value can only be popped after it is pushed. We use this fact to avoid the $O(n)$ initialization costs of the Yang-Heath scheme.

- If all values pushed to the stack are eventually popped, the values read from the stack are already a permutation of the values written, so adding "dummy operations" to the stack accesses is not required. This halves the costs of the permutation proofs.
- The Yang-Heath scheme uses a "clock" that ranges from 1 to T and is used to check that a read is reading a value written in the past. In our scheme, "dummy writes" don't have to be inserted for reads. This means the timer can only be incremented on write operations while maintaining soundness. Reducing the number of timestamps decreases the costs of the lookup table required for validating timestamps in the Yang-Heath scheme.

We now present our scheme for stack-based memory checking in Figure 2. We borrow 2 constructions from the Yang-Heath paper. We describe their costs and give a brief summary of how they work.

First, we borrow the `ro-kvs-set` primitive from Section 4.2 of the Yang-Heath paper. This is a set primitive that we will use to query whether a value belongs to the range $1...M$ for some M. Generalizing their approach, for a set primitive with M elements and N queries, their set primitive requires $M + N$ advice values, two fan-in $M + N$ multiplication gates, and $O(M + N)$ linear gates. Their read-only set construction is very similar to the plookup protocol [8].

Secondly, the Yang-Heath paper uses the syntax $\sim$ to represent a permutation check that 2 vectors permute each other. When $\overrightarrow{x}, \overrightarrow{y}$ are vectors of T elements of $\mathbb{F}$, $\overrightarrow{x} \sim \overrightarrow{y}$ checks that the root hashes of x, y with a random key are equal to each other. When they are vectors of ℓ-tuples of $\mathbb{F}$ elements, $\overrightarrow{x} \sim \overrightarrow{y}$ takes the inner product of each element of $\overrightarrow{x}, \overrightarrow{y}$ with a random vector α of ℓ values and checks that these vectors of inner products are permutations of each other via the root hash. By the properties of algebraic hashing and random linear combinations, either variant achieves negligible soundness error in checking that two vectors permute each other. Such checks have a gate cost of $2T$ multiplication gates, where T is the size of the vectors being permutation-checked.[3]

We specify the stack scheme in $Figure$ 2.

5.1. **Soundness Analysis.**

Observe that $s.depth$ always tracks the size of the stack, and is always equal to the address of the element of the stack that would be popped if the stack were read.

It remains to show that the values read from each address are consistent (meaning that each value read from an address is equal to the most recently written value) and that the stack must be nonempty at each read.

To show that the values read are consistent, we can adopt the argument of Yang and Heath, which argues that at all points throughout an execution of a sequence of stack operations, the stack satisfies an invariant: When the stack has depth m, the m corresponding entries of $s.pushes$ "lead" the entries of $s.pops$ by one entry for each valid address. This is clear for an access pattern consisting of only `push` operations on the stack. Now, whenever `pop` is executed, the depth decreases by 1. The prover must provide a tuple of $\{m, val, t'\}$ such that $t' < s.clock$. For the permutation check of $pushes, pops$ to pass with high probability, the only possible value that the prover can provide is the one value of $\{m, val, t'\}$ by which $pushes$ led $pops$ at that point in the execution, thus preserving the invariant. Likewise, `push` obviously preserves this invariant.

To show that the stack is non-empty, if the stack is popped with depth 0 for the first time at some time t, $s.pops$ must contain a tuple of $(-1, v, t')$, where v is some value, -1 is the address, and t' is a time satisfying $t' \leq t$. There cannot have been an entry in $s.pushes$ with address -1 during the first pop from address -1. The only way to add a corresponding entry to $s.pushes$ at address -1 is to perform a push at some time $t'' > t$. However, $(-1, v, t') \neq (-1, v, t'')$, so the permutation check will only pass with negligible probability.

[3]As before, taking random linear combinations of ℓ-tuples of $\mathbb{F}$ is considered a linear operation since the weights are public.

```
type Stack$_\ell$ {
  $pushes$ : record$_\ell^*$
  $pops$ : record$_\ell^*$
  $valid\text{-}diffs$ : set
  $clock$ : $\mathbb{F}$
  $depth$ : $\mathbb{N}$
}

type record$_\ell$ {
  $address$ : $\mathbb{F}$
  $value$ : $\mathbb{F}^\ell$
  $time$ : $\mathbb{F}$
}

// T is the number of conditional push operations
setup$_\ell$($T : \mathbb{N}$) $\to$ Stack$_\ell$ {
  $valid\text{-}diffs \leftarrow$ setup-set($0, \dots, T$)
  $m \leftarrow$ Stack$_\ell$ { { }, { }, $valid\text{-}diffs, 1, 0$ }
  return $m$
}
```

```
// Performs operation when g = 1, no-op otherwise
push($s$ : Stack$_\ell$, $w : \mathbb{F}^\ell$, $g : \{0,1\}$) {
  $s.depth \leftarrow s.depth + g$
  if ($g = 1$) {$s.pushes$.append({ $s.depth, w, s.clock$ })}
  $s.clock \leftarrow s.clock + g$
}
// Performs operation when g = 1, no-op otherwise
pop($s$ : Stack$_\ell$, $g : \{0,1\}$) $\to \mathbb{F}^\ell$ {
  $val \leftarrow$ input$_\ell$( )
  $t \leftarrow$ input$_1$( )
  prove-member($s.valid\text{-}diffs, s.clock - t$)
  $addr \leftarrow s.depth$
  if ($g = 1$) {$s.pops$.append({ $addr, val, t$ })}
  $s.depth \leftarrow s.depth - g$
  return $val$
}
teardown($s$ : Stack$_\ell$) {
  for $i \in [s.depth]$ {
    $val \leftarrow$ input$_\ell$( )
    $t \leftarrow$ input( )
    $s.pops$.append($\{i, val, t\}$)
  }
  $s.pushes \sim s.pops$
  teardown-set($s.valid\text{-}diffs$)
}
```

FIGURE 2. Our ZK stack scheme. Each stack slot holds an ℓ-tuple of field elements. The memory is defined in terms of the `set` data structure from the Yang-Heath paper. Valid usage consists of one call to `setup`, T calls to `push`, at most T calls to `pop` that are executed, and one call to `teardown`. All uses of nonlinear gates are highlighted.

The soundness error of the scheme is negligible, and comes from a linear error term in the permutation checks associated with `ro-kvs-check` and the error from permutation-checking the push/pop vectors.

5.2. **Scheme Costs.** Let there be T conditional push operations performed and T' conditional pop operations. Let n be the maximum depth reached by the stack. The costs of this scheme are as follows:

- $(\ell + 1) * T$ advice values as inputs for `pop` and $2T$ advice values as inputs for the set-based permutation checking used in `pop`.
- 2 fan-in $2T$ multiplication for the set membership checks. One of the fan-in $2T$ multiplications is over public values and can be optimized to require T multiplications as in the Yang-Heath paper.
- 2 fan-in T multiplications for permutation-checking $s.pushes$ and $s.pops$.
- $T + T'$ multiplications for implementing the `if` statements in `push`, `pop`. (each if statement can be implemented in 1 multiplication)
- $(\ell + 1)n$ advice values for `teardown`.
- n multiplications for `teardown`.

- $O(n + T + T')$ miscellaneous linear operations.
- $O(\ell(n + T + T'))$ operations to hash tuples of $\{i, val, t\}$ for $pushes, pops$. These are linear in our cost model. This requires $\ell + 2$ multiplications and additions per push/pop.

For a sequence of T conditional pushes and T' conditional pops (with each memory slot storing $\ell = 1$ field elements), `pop` uses $4T$ advice values. It uses $5T$ multiplications (T multiplications for permutation checking, T multiplications for the if statement, and $3T$ multiplications for the range checks). `push` uses no advice values. It uses $2T'$ multiplications (T' for the if statement and T' for the permutation checking). `teardown` requires $2n$ advice values, n multiplications, and the costs of the permutation proof/set teardown are already accounted for.

Assuming that n is negligible compared to T, T' (which makes teardown require negligibly many operations), this uses 5 multiplications and 4 advice values per push, and 2 multiplications and no advice values per pop. We feel this is a reasonable assumption since most stack-based algorithms empty their stack by the end of program execution and a program that writes a large number of unread values to the stack is inefficient and seems like a degenerate case.

Note that if the stack is not empty at the end of execution, some work can be saved in the teardown phase over popping the values by avoiding the `prove-member` function and simply adding in all required reads to $s.pushes$ directly. At the end of execution all pushes have occurred at previous clock values, so this check is unnecessary.

6. Discussion

We have demonstrated a technique for implementing queues in zero-knowledge proofs that requires 3 multiplication gates/1 advice value per read and 2 multiplication gate per write, as well as a technique for stacks that requires 5 multiplication gates/4 advice values per read and 2 multiplication gates per write. Assuming the stack/queues are read from approximately as often as they are written to, this gives an average cost of 2.5 multiplications/0.5 advice values per access for the queue, and 3.5 multiplications/2 advice values per access for the stack. These are better than the costs of 6 multiplications and 4 advice values per access for the Yang-Heath scheme for generic RAM.

These techniques are cheaper per operation in terms of constraints than generic RAM techniques, at the expense of limitations on the patterns of memory accesses. As mentioned in [14], randomness and interaction in zero-knowledge proof circuits give the ability to replace cryptographic hash functions with algebraic universal hash functions for a substantial concrete performance improvement. Our stack scheme can be thought of as replacing Reef's Poseidon-based scheme with a variant based on much cheaper universal hash functions.

While we do not have an obvious use case for the queue scheme, the stack scheme is potentially useful when implementing pushdown automata, which are used in regex matching engines like Reef [1], or in performing language parsing in zero-knowledge [13]. This would also be potentially useful in implementing a stack-based virtual machine in Zero-Knowledge proofs, such as the EVM.

7. Acknowledgements

Thank you to Alex Ozdemir, Yibin Yang and Ian Miers for their comments and help with checking my results.

References

[1] S. Angel, E. Ioannidis, E. Margolin, S. Setty, and J. Woods. Reef: Fast succinct non-interactive zero-knowledge regex proofs. Cryptology ePrint Archive, Paper 2023/1886, 2023.

[2] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems. Cryptology ePrint Archive, Paper 2012/071, 2013.

[3] J. Bootle, A. Cerulli, J. Groth, S. Jakobsen, and M. Maller. Nearly linear-time zero-knowledge proofs for correct program execution. Cryptology ePrint Archive, Paper 2018/380, 2018.

[4] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state (extended version). Cryptology ePrint Archive, Paper 2013/356, 2013.
[5] M. Campanelli, D. Fiore, S. Han, J. Kim, D. Kolonelos, and H. Oh. Succinct zero-knowledge batch proofs for set accumulators. Cryptology ePrint Archive, Paper 2021/1672, 2021.
[6] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur. Geppetto: Versatile verifiable computation. Cryptology ePrint Archive, Paper 2014/976, 2014.
[7] N. Franzese, J. Katz, S. Lu, R. Ostrovsky, X. Wang, and C. Weng. Constant-overhead zero-knowledge for RAM programs. Cryptology ePrint Archive, Paper 2021/979, 2021.
[8] A. Gabizon and Z. J. Williamson. plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Paper 2020/315, 2020.
[9] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. Cryptology ePrint Archive, Paper 2019/458, 2019.
[10] A. E. Kosba, D. Papadopoulos, C. Papamanthou, M. F. Sayed, E. Shi, and N. Triandopoulos. TrueSet: Faster verifiable set computations. Cryptology ePrint Archive, Paper 2014/160, 2014.
[11] A. Kothapalli, S. Setty, and I. Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. Cryptology ePrint Archive, Paper 2021/370, 2021.
[12] E. Laufer, A. Ozdemir, and D. Boneh. zkPi: Proving lean theorems in zero-knowledge. Cryptology ePrint Archive, Paper 2024/267, 2024.
[13] H. Malvai, S. Hussain, G. Neven, and A. Miller. Practical proofs of parsing for context-free grammars. Cryptology ePrint Archive, Paper 2024/562, 2024.
[14] A. Ozdemir, E. Laufer, and D. Boneh. Volatile and persistent memory for zkSNARKs via algebraic interactive proofs. Cryptology ePrint Archive, Paper 2024/979, 2024.
[15] S. Setty, S. Angel, T. Gupta, and J. Lee. Proving the correct execution of concurrent services in zero-knowledge. Cryptology ePrint Archive, Paper 2018/907, 2018.
[16] Y. Yang and D. Heath. Two shuffles make a RAM: Improved constant overhead zero knowledge RAM. Cryptology ePrint Archive, Paper 2023/1115, 2023.