# **stackproofs**: Private proofs of stack and contract execution using PROTO**GALAXY**

July 9, 2024

**Abstract**

The goal of this note is to describe and analyze a simplified variant of the zk-SNARK construction used in the Aztec protocol. Taking inspiration from the popular notion of Incrementally Verifiable Computation[**?**] (IVC) we define a related notion of *Repeated Computation with Global state* (RCG). As opposed to IVC, in RCG we assume the computation *does* terminate before proving starts, and in addition to the local transitions some global consistency checks of the whole computation are needed. However, we require the memory efficiency of the prover to be close to that of an IVC prover not required to prove this global consistency. We show how RCG is useful for a private smart contract system like Aztec.

## 1 Introduction

In IVC, PCD [**?**, **?**] we have an acyclic computation. However code written in almost any programming language *is* cyclic in the sense of often relying on internal calls - we start from a function $A$, execute some commands, go into a function $B$, execute its commands, and go back to $A$. When making a SNARK proof of such an execution, we typically linearize or "flatten" the cycle stemming from the internal call, in one of the following two ways

1. The monolithic circuit approach - we "inline" all internal calls (as well as loops) into one long program without jumps.

2. The VM approach - assume the code of $A, B$ is written in some prespecified instruction set. The program is executed by initially writing the code of $A, B$ into memory, and loading from memory and executing at each step the appropriate instruction according to a program counter. (For example, the call to $B$ is made by changing the counter to the first instruction of $B$.) To prove correctness of the execution, all we need is a SNARK for proving correctness of a certain number of steps of a machine with this instruction set, and some initial memory state.

The second approach is more generic, while the first offers more room for optimization, so we'd want to use it in resource-constrained settings, e.g. client-side proving.

However, what if we're in a situation where $A$ and $B$ have already been "SNARKified" separately? Namely, there is a verification key attached to each one, and we are expected to use these keys specifically. This is what happens in the Aztec system:

The Aztec private contract system: Similar to Ethereum - we have contracts; and the contracts have functions. A function in a contract can internally call a different function in the same or a different contract. Moreover, while writing the code for the different functions, we can't predict specifically what function will be internally called by a given contract function: For example, a "send token" function could have an internal call to an "authorize" function. But "authorize" is not tied to one specific verification key - as different token holders are allowed to set their own "authorize" function.

The goal of the Aztec system is to enable constructing zero-knowledge proofs of such contract function executions. For this purpose, a contract is deployed by

1. Computing a verification key for each function of the contract.
2. Adding a commitment to the verification keys of the contract in a global "function tree'. More accurately, a leaf of this tree is a hash of the contract address with a merkle root of a tree whose leaves are the verification keys of the contract functions.

While running, a contract can read, create or delete notes belonging to the contract. We can thus think of the notes as "global variables" shared between the different functions.

Let us think of all functions in this system as having multiple arguments, and returning accept or reject. (We can always move the output into the arguments, if it's not of this form.) Here's a natural way to prove the mentioned execution: Put the arguments to $B$ in the public inputs of both the circuits of $A$ and $B$. Verify the proofs $\pi_A, \pi_B$ for $A, B$; and check via the public inputs the same value was used in both proofs for the arguments of $B$.

However, this doesn't yet deal with the notes. During native execution, note operations happened at a certain order. We can think of these operations as having timestamps incremented by one with each operation. We need to check, for example, that if a note was read in a certain timestamp, it was indeed created in an earlier timestamp. We can have the note operations - $\{add, read, delete\}$ - performed by a function be included in the public inputs of its circuit. The issue is, what if $A$ is reading a note that was created in the internal call to $B$?

This brings us to the notion of *Repeated Computation with Global state* (RCG). In RCG we have a transition predicate taking us from one state to the next. We wish to prove we know a sequence of witnesses taking us from a legal initial state to a certain publicly known final state. This might remind the reader of the popular notion of *incrementally verifiable computation* (IVC). There are two differences.

- In RCG we are not interested in "incremental" proofs of one step, only in proofs for a whole sequence of transitions ending in a desired final state.

- In RCG we allow a *final predicate* checking a joint consistency condition between witnesses from all iterations.

From what is written so far, one could ask why not *only* have a final predicate that includes the transition checks. The point is that in our usecase the final predicate is applied to small parts of each witness - namely the note opreations, and is relatively simple. Thus, this decomposition facilitates obtaining better prover efficiency, compared to what we'd get from a monolithic circuit for the whole computation.

# 2 Preliminaries

## 2.1 Relations of the app functions

Define a relation including the selectors Fixed polynomial $f(x_1, \ldots, x_S)$ $S = n + d + \ell$ Relation $\mathcal{R}_{app}$ of $\mathsf{cm1}, \ldots, \mathsf{cm}_S$

# 3 Repeated Computation with Global state

An RCG relation is defined by a pair of functions $(F, f)$.
We call $F(Z, W, S, Z^*) \rightarrow \{\mathsf{acc}, \mathsf{rej}\}$ the *transition predicate*.
We informally think of

- $Z^*$ as the output of $F$ (although the actual output is $\{\mathsf{acc}, \mathsf{rej}\}$.
- $Z$ as the public input and $W$ as the private input of $F$.
- $S$ as the part of the private input that will be used in the final predicate.

$f : D^* \times V \rightarrow \{\mathsf{acc}, \mathsf{rej}\}$ is the *final predicate*.

The relation $\mathcal{R} = \mathcal{R}_{F,f}$ consists of pairs $(\mathsf{x}, \omega)$ such that $\mathsf{x} = (z_{\text{final}}, C, V), \omega = (n, z = (z_0, \ldots, z_n), w = (w_1 \ldots, w_n), s = (s_1, \ldots, s_n))$ such that

- $z_0.\mathsf{init} = \mathsf{true}$.
- $z_n = z_{\text{final}}$.
- $n \leq C$.
- For each $i \in [n]$, $F(z_{i-1}, w_i, s_i, z_i) = \mathsf{acc}$.
- $f(s_1, \ldots, s_n, V) = \mathsf{acc}$.

we say a zk-SNARK for $\mathcal{R}$ is *space-efficient* if given $s$ and streaming access to $z$ and $w$ $\mathbf{P}$ requires space $O(|F| + |V|)$.

# 4 Record operations

*Records* are pairs $(v, c)$ - $v$ is a value, $c$ is a counter. A record operation has one of the following forms $(\mathsf{add}, v, c)$,$(\mathsf{del}, v, vc, c)$,$(\mathsf{read}, v, vc, c)$.

Where $v$ denotes a note value, $c$ denotes the counter of the operation. For read/del - $vc$ denotes the counter where the value being read or deleted

We say a sequence $O$ of record ops of size $n$ is *consistent* if - The *count* fields in all ops are distinct, and as a set equal to $\{1, \ldots, n\}$. - The *valc* fields in all *del* ops are distinct. - If $(val, \mathsf{read}, valc, c) \in ops$, then $valc < c$ and $(val, \mathsf{add}, valc) \in ops$. - If $(val, \mathsf{del}, valc, c) \in ops$ then $valc < c$ and $(val, \mathsf{add}, valc) \in ops$.

Let $V$ be set of records. We say $O$ is *consistent with* $V$ if:

- $O$ is consistent.
- $V = \{(val, c) | (\mathsf{add}, val, c) \in O \;\&\; \forall c', (\mathsf{del}, val, c, c') \notin O\}$. In words, $V$ is the set of notes that were added but not deleted.

## 4.1 Proving record ops via log-derivative

# 5 The transition function $F$ describing the Aztec kernel

Here's a sketch of what the Aztec $F$ could look like:

Global state: - $r_{func}$ - merkle tree of allowed functions

Outputs: - $g$ - stack of functions - where each function consists of its selector commitments - $ns$ - new state to be added to Aztec - notes and nullifiers. To be of fixed size, $ns$ might be represented as a merkle root.

Public inputs - $g_{prev}$ - previous stack of functions - $ns_{prev}$ - previous new state to be added to Aztec - notes and nullifiers

Private inputs: - $pi_f$ - public inputs to some Aztec function $f$. These include - Descriptions of at most four functions $f_1, \ldots, f_4$ to add to the stack. Again, a description of a function is its selector commitments. - $w_f$ - the witness of the Aztec function $f$ to be executed. - $i$ - index of $f$ in the Aztec function tree.

For boolean $init$, global state $r_{func}$, private input $\omega = (pi_f, w_f, i, aux)$, public input $z = (g_{prev}, ns_{prev})$, and output $z^* = (g, ns)$, we have $F(init, g, z, w, z^*) = acc$ exactly when:

- Let $f$ be the first function in $g_{prev}$. Then - $f$ is contained at leaf $i$ in the tree with root $r_{func}$ - $w_f$ is a valid witness for $f$ with public inputs $pi_f$ (There are subtleties in how this is done, see below) - adding $ns_f$ to $ns_{prev}$ results in $ns$. - let $f_1 \ldots, f_4$ be the functions in $pi_f$. Then popping $f$ from $g_{prev}$ and pushing $f_1, \ldots, f_4$ results in $g$. - If $init = true$ then $g_{prev}$ contains exactly one function, i.e. $f$; and $ns_{prev}$ is empty. - $aux$ contains all merkle paths necessary in the above.

**Our $z_{final}$:**

$z_{final}$ must contain empty stack $g$, and $ns$ matching the notes and nullifiers we send to the rollup

# 6 On protogalaxy

## 6.1 Introspective constraints

Constraints $f_i$ on $\omega$, simply low degree polynomials, but have the ability to refer to components of $\mathsf{cm}(m_j)$ Function $F$ should have "introspection" ability to look at commitments.

commitment function will output two representations of $\mathsf{cm}(w)$ - in $\mathbb{G}$ and in $\mathbb{F}$ and

or PI will include $\mathbb{F}$ representation. which will then be part of $w$. **V** will check representations match.

## 6.2 Proving ProtoGalaxy under zero-testing assumption

# 7 The Algebraic Group Model with recursive extraction

As in [FKL18], we assume when $\mathcal{A}$ outputs $a \in \mathbb{G}$ it outputs a vector $c \in \mathbb{F}^n$ with $< c, \mathsf{srs} >= a$. We fix some mapping $G : \mathbb{F}^4 \to \mathbb{G} \cup \{*\}$. $*$ means the input doesn't correspond to $\mathbb{G}$ element. We assume that this is the representation $\mathcal{A}$ uses for elements of $\mathbb{G}$.

For our security proof, we require a notion of "recursive extraction" used by [**?**]: Specifically, we assume that if $c$ output by $\mathcal{A}$ along with $a$, thought of as $c \in (\mathbb{F}^4)^{n/4}$, contains an element $c_i$ with $G(c_i) = a \in \mathbb{G}$, then $\mathcal{A}$ outputs $c' \in \mathbb{F}^n$ with $< c', \mathsf{srs} >= c$. The same holds for $c'$. Thus $\mathcal{A}$ must continue outputting representations until reaching one where no element correponds to $a \in \mathbb{G}$.

# 8 Transforming $F$ into $F'$

Describing $F^*$ **public input:** $z$ - output for $F$ $G$ - global state for $F$ $count$ - counter of IVC step $h$ - supposed hash of accumulator **private input:** $acc$ - current accumulator instance $acc_{prev}$ - previous accumulator instance $inst$ - instance (of $F'$) to be accumulated. $w$ - private input for $F$ $\pi$ - proof for protogalaxy verifier

Set $X := (z, G, count, h)$, $W := (acc, acc_{prev}, inst, w, \pi)$

$F'(X, W) =$ accept if and only if:

1. $hash(acc) = h$.
2. $V_{PG}(acc_{prev}, inst, \pi, acc) =$ acc
3. $inst.h = hash(acc_{prev})$.
4. If $count > 0$:
   (a) $inst.G = G$.
   (b) $inst.count = count - 1$.
   (c) $F(false, G, inst.z, w, z) =$ acc

5. If $count = 0$:

   (a) $F(true, G, inst.z, w, z) = \mathrm{acc}$

## Acknowledgements

## References

[FKL18] G. Fuchsbauer, E. Kiltz, and J. Loss. The algebraic group model and its applications. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II*, pages 33–62, 2018.