From IVCs to RCGs

Ariel Gabizon

Aztec Labs

Outline

- The Aztec Smart Contract system
- RCG
- Global state via log derivative
- A theoretical issue

The Aztec Private Smart Contract System

A *contract* has functions - represented by *verification keys* .

$\boldsymbol{A} - \mathsf{\mathbf{vk}}_{A}$
$\mathbf{B} - \mathsf{\mathbf{vk}}_{\mathbf{B}}$

The Aztec Private Smart Contract System

A *contract* has functions - represented by *verification keys* .

$\mathbf{A} - \mathsf{vk}_A$
$\mathbf{B} - \mathsf{vk}_\mathbf{B}$

Function contracts can

- call other functions in same/other contract

How do circuits “call each other”?

How do circuits "call each other"?

Example: Want to prove execution of

$\boldsymbol{A}(\mathbf{args}_{A})\{$

..

..

$\mathbf{B}(\mathbf{args}_{B});$

..

..

$\}$

How do circuits "call each other"?

Example: Want to prove execution of

$\boldsymbol{A}(\textbf{args}_{\mathbf{A}})\{$

..

..

$\mathbf{B}(\textbf{args}_{\mathbf{B}});$

..

..

$\}$

Idea: $\mathbf{A}$*'s public input will contain* $\textbf{vk}_{\mathbf{B}}$ *and* $\textbf{args}_{\mathbf{B}}$

How do circuits "call each other"?

Construct proofs -

- π_A for A with public input $x_A = (\mathbf{args}_A, \mathbf{vk}_B, \mathbf{args}_B)$

How do circuits "call each other"?

Construct proofs -

- π_A for A with public input $x_A = (\mathbf{args}_A, \mathbf{vk}_B, \mathbf{args}_B)$
- π_B for B with public input $x_B = (\mathbf{args}_B)$

How do circuits "call each other"?

Construct proofs -

- π_A for A with public input $x_A = (\mathbf{args}_A, \mathbf{vk}_B, \mathbf{args}_B)$
- π_B for B with public input $x_B = (\mathbf{args}_B)$

$\mathcal{V}$ checks

- (x_A, π_A) with $\mathbf{vk}_A$
- (x_B, π_B) with $\mathbf{vk}_B$

How do circuits "call each other"?

Construct proofs -

- π_A for A with public input $x_A = (\mathbf{args}_A, \mathbf{vk}_B, \mathbf{args}_B)$
- π_B for B with public input $x_B = (\mathbf{args}_B)$

$\mathcal{V}$ checks

- (x_A, π_A) with $\mathbf{vk}_A$
- (x_B, π_B) with $\mathbf{vk}_B$

As $\mathcal{V}$ enforces $\mathbf{args}_B, \mathbf{vk}_B$ used are the same in both checks - corresponds to A "calling" B.

Casting as IVC of a fixed function:

$\underline{F(\mathbf{vk}_{\mathbf{cur}}, x = (\mathbf{args}_{\mathbf{cur}}, \mathbf{vk}_{\mathbf{next}}, \mathbf{args}_{\mathbf{next}}), \pi, \mathbf{stack})}$ **:**

Casting as IVC of a fixed function:

$\underline{F(\textbf{vk}_{\textbf{cur}}, x = (\textbf{args}_{\textbf{cur}}, \textbf{vk}_{\textbf{next}}, \textbf{args}_{\textbf{next}}), \pi, \textbf{stack})}$ **:**

1. Check $(\textbf{vk}_{\textbf{cur}}, \textbf{args}_{\textbf{cur}})$ is top element in **stack** and pop it off.

Casting as IVC of a fixed function:

$F(\mathsf{vk}_{\mathsf{cur}}, x = (\mathsf{args}_{\mathsf{cur}}, \mathsf{vk}_{\mathsf{next}}, \mathsf{args}_{\mathsf{next}}), \pi, \mathsf{stack})$:

1. Check $(\mathsf{vk}_{\mathsf{cur}}, \mathsf{args}_{\mathsf{cur}})$ is top element in **stack** and pop it off.
2. Check that $\mathcal{V}(\mathsf{vk}_{\mathsf{cur}}, x, \pi) = \mathbf{acc}$.

Casting as IVC of a fixed function:

$F(\mathbf{vk}_{\mathbf{cur}}, x = (\mathbf{args}_{\mathbf{cur}}, \mathbf{vk}_{\mathbf{next}}, \mathbf{args}_{\mathbf{next}}), \pi, \mathbf{stack})$:

1. Check $(\mathbf{vk}_{\mathbf{cur}}, \mathbf{args}_{\mathbf{cur}})$ is top element in $\mathbf{stack}$ and pop it off.
2. Check that $\mathcal{V}(\mathbf{vk}_{\mathbf{cur}}, x, \pi) = \mathbf{acc}$.
3. Push $(\mathbf{vk}_{\mathbf{next}}, \mathbf{args}_{\mathbf{next}})$ to top of $\mathbf{stack}$.

But we forgot global state

A *contract* has functions - represented by *verification keys* .

$\mathbf{A} - \mathsf{vk}_A$
$\mathbf{B} - \mathsf{vk}_\mathrm{B}$

But we forgot global state

A *contract* has functions - represented by *verification keys* .

$\mathbf{A} - \mathbf{vk}_A$
$\mathbf{B} - \mathbf{vk}_{\mathrm{B}}$

and *notes* representing its state:
$\mathbf{note}_1$
$\mathbf{note}_2$

But we forgot global state

A *contract* has functions - represented by *verification keys* .

$A - \mathbf{vk}_A$
$B - \mathbf{vk}_B$

and *notes* representing its state:
$\mathbf{note}_1$
$\mathbf{note}_2$

Function contracts can

- call other functions in same/other contract
- add/read/delete contract notes

Global State

We add to the function public inputs the note operations (with time stamps)

$x_A = (\mathbf{args}_A, \mathbf{vk}_B, \mathbf{args}_B, [\mathbf{read}, \mathbf{note}, \mathbf{8}])$

$x_B = (\mathbf{vk}_B, \mathbf{args}_B, [\mathbf{add}, \mathbf{note}, \mathbf{3}])$

Problem: When verifying proof for $\boldsymbol{A}$ we don't know whether in a future IVC iteration we'll see **note**created (with earlier timestamp)

Global State

We add to the function public inputs the note operations (with time stamps)

$x_{\mathrm{A}} = (\mathbf{args}_{\mathrm{A}}, \mathbf{vk}_{\mathrm{B}}, \mathbf{args}_{\mathrm{B}}, [\mathbf{read}, \mathbf{note}, \mathbf{8}])$

$x_{\mathrm{B}} = (\mathbf{vk}_{\mathrm{B}}, \mathbf{args}_{\mathrm{B}}, [\mathbf{add}, \mathbf{note}, \mathbf{3}])$

Problem: When verifying proof for $\boldsymbol{A}$ we don't know whether in a future IVC iteration we'll see **note**created (with earlier timestamp)

"Order of proving is different than order of execution"

RCG - Repeated Computation with Global state

Like IVC...but

- Computation ends before proving starts.
- Prover memory allowed to depend on size of global state in addition to memory for one iteration

RCG - Repeated Computation with Global state

Like IVC...but

- Computation ends before proving starts.
- Prover memory allowed to depend on size of global state in addition to memory for one iteration

RCG - Simplfied dfn

Transition predicate: $F(Z, W, Z^*, S) \to \{\mathbf{acc}, \mathbf{rej}\}$.
Final predicate: $f(S_1, \ldots, S_n) \to \{\mathbf{acc}, \mathbf{rej}\}$.
The RCG relation $\mathcal{R} = \mathcal{R}_{F,f}$ consists of pairs (X, W) such that $X = (z_{\mathsf{final}}, n), W = (z = (z_0, \ldots, z_n), w = (w_1 \ldots, w_n), s = (s_1, \ldots, s_n))$ such that

- $z_n = z_{\mathsf{final}}$.
- For each $i \in [n]$, $F(z_{i-1}, w_i, z_i, s_i) = \mathbf{acc}$.
- $f(s_1, \ldots, s_n) = \mathbf{acc}$.

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

Déjà vu from previous talk: Memory checks with log-derivative [Eagen22, Haböck22]

. In **add** ops also write the number of times note is read e.g. $\alpha = (\mathbf{add}, \mathbf{note}, \mathbf{numreads})$ In **read** ops write the timestamp of note addition. e.g. $r = (\mathbf{read}, \mathbf{note}, \mathbf{acount}, \mathbf{cnt})$.

- For each read r we check that $\mathbf{acount} < \mathbf{cnt}$.
- Prover hashes note operations from all function calls to get challenge $\beta \in \mathbb{F}$.
- Final proof will check that

$$\sum_{r \in \mathbf{reads}} \frac{\mathbf{1}}{\mathbf{note} + \beta \cdot \mathbf{acount}} = \sum_{\alpha \in \mathbf{adds}} \frac{\mathbf{numreads}}{\mathbf{note} + \beta \cdot \mathbf{cnt}}$$

Theoretical interlude - The recursive Algebraic Model [LS23]

AGM[FKL] - Given SRS $\boldsymbol{v} \in \mathbb{G}^n$, when $\boldsymbol{\mathcal{A}}$ outputs $\mathbf{a} \in \mathbb{G}$ it must output $\mathbf{c} \in \mathbb{F}^n$ such that $\mathbf{a} = \sum_{i=1}^{n} \mathbf{c}_i \boldsymbol{v}_i$.
Fix in advance representation function for $\mathbb{G}$ $\mathbf{repr} : \mathbb{G} \longrightarrow \mathbb{F}^2$. What if for some $\mathbf{i} < \mathbf{n}$, $(\mathbf{c}_i, \mathbf{c}_{i+1}) = \mathbf{repr}(\mathbf{b})$ for some $\mathbf{b} \in \mathbb{G}$? Then a *recursive Algebraic adversary* must output $\mathbf{c}' \in \mathbb{F}^n$ with $\mathbf{b} = \sum_{i=1}^{n} \mathbf{c}'_i \boldsymbol{v}_i$.

For more details see:

stackproofs: Private proofs of stack and contract execution using Protogalaxy

Liam Eagen[1], Ariel Gabizon[2], Marek Sefranek[3], Patrick Towa[2], and Zachary J. Williamson[2]

[1]Alpen Labs
[2]Aztec Labs
[3]TU Wien

October 5, 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[Val08] (IVC) we define a related notion of *Repeated Computation with Global state* (RCG). As opposed to IVC, in RCG we assume the computation terminates before proving starts, and in addition to the local transitions some global consistency checks of the whole computation are allowed. However, we require the space 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 designing a proof system for a private smart contract system like Aztec.

1 Introduction

Incrementally Verifiable Computation (IVC) [Val08] and its generalization to Proof Carrying Data (PCD) [CT10] are useful tools for constructing space-efficient SNARK provers[BCCT12]. In IVC and PCD we always 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