CS 189A Project Requirements Document Version 1

Company: Veridise
Team: Logos (Team 9)
Project: ZK Circuit IR Generator
Team members:

- Shiv Kapoor - Lead
- Thomas Hale - Scribe
- Colter Sirlin
- Chris Chang
- Maya Ma

Background:

Blockchains are the embodiments of tamper-proof data structures that are highly accessible at all times. Due to their utility in enforcing critical agreements that can carry high amounts of monetary value, block space has become commoditized and expensive. Many methods of increasing block space have been proposed, one of the most popular ones being rollups. Rollups involve rolling up or batching transactions using an off-chain virtual machine (sequencer) and periodically submitting the off-chain state changes to the blockchain. Rollups come in two different flavors: optimistic and zk. The former employs an approach which assumes that all off-chain transactions are valid unless a party submits a fraud proof which invalidates an off-chain state transition. The latter, on the other hand, leverages Zero-Knowledge circuits and proofs to prove the validity of each state transition that occurred off-chain. Moreover, these circuits are mission critical to ensuring the integrity of rollup transactions and must be carefully designed, written, and audited in order to avoid insidious flaws which could compromise user's funds and data. The proof is generated by the rollup sequencer and submitted to an on-chain contract which verifies the proof and applies the underlying state transitions.

There are two varieties of Zero-Knowledge proofs: zk-SNARKs (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge) and zk-STARKs (Zero-Knowledge Scalable Transparent Arguments of Knowledge). They are cryptographic protocols that enable one party to prove to another that a given statement is true without revealing any specific information about the statement itself, except for its validity. zk-SNARKs, particularly popular in the blockchain space, require a trusted setup but are computationally more efficient. In contrast, zk-STARKs do not require a trusted setup and are transparent but tend to be computationally heavier. Both are pivotal in ensuring privacy and scalability in various cryptographic and blockchain applications.

The benefits provided by zk-SNARKS and zk-STARKS can only be realized if the ZK circuits themselves are secure. Detecting and rectifying vulnerabilities in ZK circuits is paramount, as even a minor flaw can lead to significant breaches, undermining the trust and reliability that blockchains aim to establish. Veridise specializes in auditing ZK circuits and smart contracts to ensure their security and robustness by examining them for potential vulnerabilities.

However, a challenge for Veridise is the number of frameworks ZK circuits are written in. Each framework has its own structure and nuances, making it challenging to develop universal security analysis tools.

Goals/Expected Outcomes

Our primary objective is to create an intermediate representation (IR) capable of capturing the behavior of Zero-Knowledge (ZK) circuits from various frameworks. This involves defining the IR and creating systems to transpile ZK circuits written in a supported framework into our IR. Transpilation will work on the user’s local machine.

Once we have the IR and working translation, we aim to design a user-friendly visualization tool for ZK circuits using our IR.

Additionally, we aim to package transpilation and visualization tools into a publicly accessible web application for easy interaction and conversion.

As a stretch goal, we aspire to implement an interactive hover feature in the visualization tool, allowing users to highlight converted expressions upon hovering over specific elements.

Implementation Choices

IR Design
A good IR will capture the program flow and computational elements that compose a ZK circuit. We initially considered building a compiler to directly translate frameworks into an IR. However, frameworks are written in full programming languages such as Rust or

Go, so this would be akin to building a full compiler for a general-purpose language. However, there is an easier way to build an IR.

We can use a technique we are calling “emitting an IR as a side effect.” We can “piggyback” on the existing compilers for these languages by passing an object through the framework library functions called by the circuit. Each function will modify the object so that its execution is recorded. This will build up a graph or tree in the syntax of our IR that represents the circuit. This can be achieved by forking the framework from GitHub and modifying it so that this information can be recorded on the side.

Languages Used

We plan to start with Plonky2, which is written in Rust. This means we will write our forked implementation of Plonky2 in Rust. The IR itself will be in JSON format. This seems like a natural way to represent the hierarchical structure of ZK circuits. JSON is also widely used and many tools and programming languages support it. This means we can easily work with our IR in other languages if needed. Additionally, the JSON format will also make it rather simple to render a graph representation of the ZK circuit in our web application.

We plan to run transpilation (conversion from ZK framework to our IR) locally using webassembly, or on the cloud using AWS Lambda and API Gateway. The web app itself will use React to display a graphical representation of the JSON IR.

User Interaction and Design

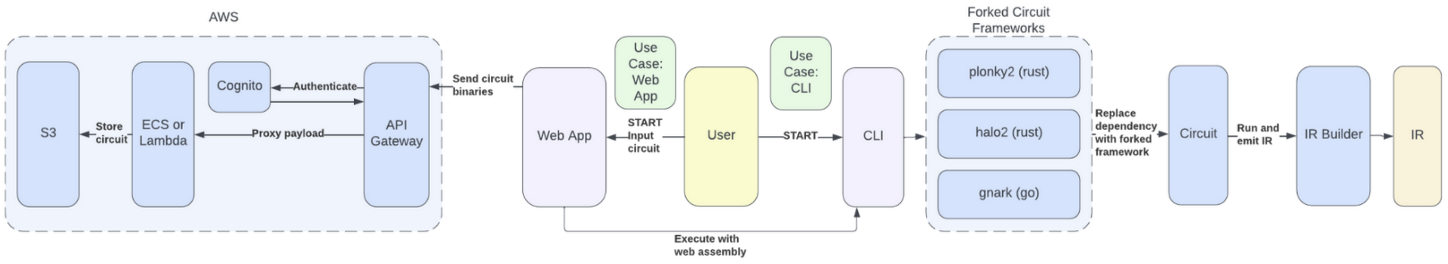
The user will interact with our service using our web application. They can provide their ZK circuit by directly uploading source code from a supported framework, or by submitting a link to a git repository. Then, our Logos IR generator will convert the source code into its intermediate representation and return it to the user as a JSON file. The JSON file will also be rendered as a graph so the user has another modality for interpreting the intermediate representation.

If we are able to achieve the objectives delineated above, we will also look into integrating a LLM to perform security analysis on the intermediate representation and deliver an AI generated audit report for the ZK circuit.

User Scenario: command line interface (CLI) - replaces circuit frameworks with our fork and runs circuit. As the circuit runs, our fork bookkeeps operations and constraints and emits the IR as a side effect

User Scenario: Web App - runs the CLI’s core through webassembly. It also sends the input file binaries to AWS api gateway, which authenticates with Cognito. Then it proxies the payload to ECS or lambda depending on the target scalability of our cloud infrastructure. Our ECS or lambda will then store the binaries to S3. This enables features like applying AI models onto the inputs or retrieving user history of circuits.

Architecture



User Stories

As a user, I want to be able to convert my plonky2 circuit to an IR

- Plonky2 is supported by Logos IR generator

As a user, I want to be able to convert my halo2 circuit to an IR

- Halo2 is supported by Logos IR generator

As a user, I want to be able to convert my gnark circuit to an IR

- Gnark is supported by Logos IR generator

As a user, I want to be able to convert my Artworks circuit to an IR (stretch goal)

- Artworks is supported by Logos IR generator

As a user, I want to be able to compare and contrast the IR representation of circuits written in multiple different frameworks and have a cohesive environment for security analysis

- There is only a single IR
- The IR supports more than one framework

As a user, I want to generate the IR representation of my ZK circuit via a CLI so I can translate circuits locally and with minimal overhead.

- Transpilation works locally
- Transpilation can be started using a CLI command
- Transpilation results can be viewed using a CLI command

As a user, I want to generate the IR representation of my ZK circuit through a website so I can translate circuits without using my local machine's resources

- Transpilation can be hosted on a public website

As a user of the website, I want quick ways to submit a circuit for transpilation that align with my existing workflow

- There is an input to paste code from the user's files
- There is an upload file option to submit code directly
- There is an option to pull code from a GitHub repository

As a user, I want to see the transpiled code quickly after inputting, so that I can interact with the system in real time without waiting and disengaging

- The transpilation time is at most 30 seconds, and ideally less than 5 seconds

As a user, I want to be able to see the history of my previous circuits and IRs so I have a record of past conversions.

- User can sign in and sign out
- There is a table showing history of circuits
- Version Control

As a user, I want to be able to get the IR representation of my circuit in a commonly used notation like JSON

- The IR is represented in JSON

As a Veridise employee, I want to be able to get the common IR representation of a customer's circuit

- The IR is useable by Veridise for security analysis and verification

As a user, I want to have my resulting computations accessible from other machines while staying secure (stretch goal)

- Website has user accounts
- Website has secure login

Open Questions

How to convert JSON to graphical representation?
How to pull circuits from a private GitHub repo?

Appendix

Technologies Used

- Rust
- JSON
- AWS
- WebAssembly
- React
- Plonky2
- Halo2
- Gnark
- Arkworks