Keyless Blockchain

Alex Zhao, Kshitij Sodani, Tony Wu, Thomas Liu

May 2025

§1 Introduction

Traditional blockchain wallets rely on a long-term secret key (SK) that the user must safely store and manage. Losing this SK is irrevocable: the user will permanently lose access to their funds, and there's no recovery path. Worse, every dApp uses its own keypair, so users are forced to juggle multiple wallets or import buried private keys across apps, making handling for the wallet an error-prone, cross-device hazard.

To resolve this problem, Keyless accounts designate authentication to Google, averting the need for memorizing a secret key. Along the implementation, cryptographic considerations of privacy and security from third parties motivates the layering with ZK proofs of knowledge and added randomness.

Taken inspiration from the Aptos zero knowledge keyless protocol [1], we made our own implementation to understand the circuit and survey the optimization and security in hopes of improving upon it. Our implementation resulted in a simplified version of the Aptos protocol that retained the structure and altered some internal procedures.

We further analyzed the security of our protocol and discovered an attack that violates the statistically binding requirement of a ZK proof, however concluded that the security was still computationally binding.

In future work, we intend to resolve this vulnerability while still retaining the simplicity of the circuit, and automate the ZK proofs and Google authentication to deploy with other services. With a completed and secure circuit, we intend to deploy on Ethereum contracts to test the system in practice.

§2 Protocol

Since our implementation is based on Aptos keyless, we first describe the Aptos protocol before moving to our implementation. The main differences in our implementation is locally computing the zero knowledge proofs as well as implementing a different zero knowledge protocol for cheaper verification.

A high level overview of the Aptos protocol flow can be found in Figure 1. We elaborate on the technical details below.

§2.1 Key Generation

This is the first step of the protocol. The user client generates an ephemeral public/private key pair (epk, esk) for digital signing. These keys will be used to sign transactions and verify that all transactions are user approved. The ability to generate new keypairs allows the user to not store keys locally, but we need to somehow verify that the keypairs were actually generated by the user. This will be done through the Google OAuth service.

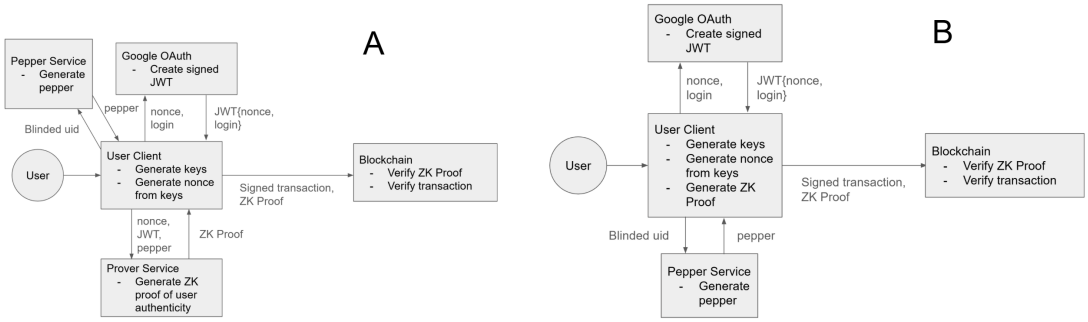

Figure 1: Flow chart A describes the flow of the Aptos protocol. Flow Chart B describes the flow of our implementation.

The user client will also randomly generate an ephemeral blinding key r which will be used to hide sensitive information from the external services.

§2.2 Google OAuth Signing

After generating keys, the user will login to their Google account through Google OAuth while using a hash of epk and r, $H(epk, r)$, as the nonce. Google will then return a signed JWT containing the nonce, user email, and expiration to the user. This signed JWT proves that a user with the corresponding Google account did in fact generate epk while also not revealing epk to Google.

§2.3 Pepper Service

Another issue with using ephemeral keys instead of permanent ones is that every user needs to have a permanent address that can send and receive transactions. However, if we have the address simply correspond to the user email, then the anonymity of the blockchain is broken. Thus, we require a pepper service to calculate a random pepper p for each user which we can then hash with the user email and app id to compute their address, $H(uid, aud, p)$

To ensure that transactions with the pepper service are zero knowledge, we will use the following discrete log based protocol:

Algorithm 1 — Pepper service discrete log protocol.

1. Pepper service chooses a prime modulus M, a prime generator g, and a random secret key sk. Then, they publish M, g, and a public key $pk = g^{sk}$
2. User generates a random blinding key u and a hash of their user email E, and computes $h = g^u E$.
3. The user sends h to the pepper service, and the pepper service will return h^{sk} to the user.
4. The user can compute their pepper $p = E^{sk}$ by evaluating $\frac{h^{sk}}{pk^u}$ as seen in Equation 1.

$$\frac{h^{sk}}{pk^u} = \frac{(g^u E)^{sk}}{(g^{sk})^u} = E^{sk} \tag{1}$$

To ensure that only the user is able to use the pepper service to retrieve their pepper, we can have the user generate a zero knowledge proof that with Google's public key and epk as public inputs, they know inputs of a signed JWT and r such that the signed JWT corresponds to Google's public key and it's nonce is equal to $H(epk, r)$. Then, the pepper service only needs to verify the proof and check that the user's epk is not expired before performing the pepper calculations.

§2.4 Prover Service

The Aptos protocol decides to have a separate service for generating zero knowledge proofs to reduce user computation. This prover service received a signed JWT, r, and p and generates a zero knowledge proof that they know these values as private inputs given the address, epk, and Google's public key as public inputs. Specifically, the proof verifies that the JWT is signed by google, the nonce is equal to $H(epk, r)$, and the address is equal to $H(uid, aud, p)$.

§2.5 Blockchain

After receiving the zero knowledge proof, the user client can send it to the blockchain and begin sending transactions. For every transaction, they sign it using their esk and send it to the blockchain. This way, the blockchain only needs to verify the zero knowledge proof once per session instead of once per transaction, because once the proof is verified the chain knows that the (epk, esk) key pair is valid.

§2.6 Our Implementation

For our implementation, we decided to move the prover service locally, since only one zero knowledge proof needs to be computed per session and there may be security concerns with a 3rd party prover service which will know both your user email and blockchain address.

For each part of the protocol, some implementation details are provided:

- The (epk, esk) keypair is generated using ed25519-dalek which generates a Ed25519 signature scheme over the Curve25519 elliptic curve.
- The blinding key r is generated as a random 256-bit integer.
- For all hashing, $H(...)$ is performed through a Sha256 hash.
- The pepper service uses a modulus $M = 10^9 + 7$ and generator $g = 3$. This is clearly not secure for modern systems, but these values can be easily changed and were only chosen to test the system. In practice we would be using values on the order of 256 bits, so after receiving the pepper, the user client performs a Sha256 hash on it to simulate having received a 256 bit pepper.
- For our zero knowledge proof protocol, we use the OpenVM implementation for proving arbitrary rust code. More technical details on this protocol can be found in the next section.

§2.7 Security

Our implementation is secure as follows:

- When a user logs into the app, Google knows that they own an address on the blockchain but they are unable to determine the address due to the user's blinding key r and the security of SHA256.
- When a user requests a pepper from the pepper service, the pepper service does not know which user has requested the pepper. The service is able to determine an address given the user id, but is unable to reverse engineer a user id from address due to the security of SHA256 and the space of user ids being too large to search.
- The security of the user's transaction signing is guaranteed by ed25519-dalek.
- The security of Google's JWT signing is guaranteed by RSA256.
- The security of the pepper service is guaranteed by the hardness of the discrete log problem. Note that we can improve the security by using a discrete log elliptic curve with the same protocol.

While working on the project, we discovered an attack for our implementation that reveals some statistical information about the user id from the zero knowledge proof. More details on the attack are provided in the next section.

§3 ZK Protocol

Our zero-knowledge protocol is implemented as a custom zkVM circuit built on the OpenVM framework, an open-source zkVM released by Axiom. The use of a zkVM significantly reduces our development overhead, since we can simply write intuitive Rust that reads "input" and computes "assertions" that compile into an execution trace in the PLONK (specifically plonky3) ZK protocol.

Internally, the circuit takes as private witnesses the Google-signed JWT (including its RSA signature), the commitment nonce for the JWT $H(\text{epk}, r)$, and all blinding factors used in the OPRF and VRF steps. Externally, however, it reveals only three public outputs: the derived on-chain public key

$$\text{pubkey} = H(sub,\ aud,\ r),$$

the ephemeral public key epk, and a session expiration timestamp. By exposing only these values, the circuit binds a fresh blockchain identity (with the corresponding ephemeral key and timestamp) to a valid Google login and pepper randomness without ever leaking the underlying secrets.

Inside the circuit, we first parse the JWT header and payload as byte arrays and verify their validity with string operations.

Next, verify the RSA signature under Google's known public key against the payload fields $\{sub,\ aud,\ nonce = H(epk, r)\}$. To do this, we had to use U2048 in Rust and pass in sub-quotients as witnesses, because the zkVM was not optimized for U2048 division.

Next, we verify the nonce $nonce = H(\text{epk}, r)$.

Finally, the circuit recomputes the address hash $H(sub, aud, r)$ and emits it as a public output, thus cryptographically linking the Google credentials and pepper to the on-chain account.

The zkVM is able to execute quickly because of precompiles for the SHA256 operation. Because of the iterative nature of SHA256, it is well-suited to proving with a PLONK-like ZK system.

We employ the Groth16 proving system via the OpenVM backend: each AIR component's execution trace is committed through low-degree extensions and Merkle trees; Fiat–Shamir challenges inject non-malleable randomness for sumchecks; and all polynomial openings are batched into a single FRI-based proof. The resulting proof π_{snark} is compact enough to verify on-chain with approximately 200 k gas, after which per-transaction authentication is reduced to a lightweight ECDSA signature under the session key esk.

Due to the nature of this compilation and of PLONK, the resulting ZK proof is not guaranteed to be statistically hiding like the original Aptos Keyless Circom proof is; however, it computationally hides all inputs. The commitment for the entire program is a Merkle root of the low-degree extension of the columns of the trace and the Merkle root of the shared RAM. Since the commitment is opened with FRI, with overwhelming probability, the actual rows of the trace itself are not revealed. Furthermore, the proof posted to the EVM contains a Halo2 proof of the FRI verification protocol itself, with the hash of the program code as an input.

§4 Specific Implementation Details

Smaller implementation details not related to security.

§4.1 User Client

The implementation for the user client can be found in [3]. The site is hosted on https://localhost:8080 and is run using Python and Flask. It handles key generation, OAuth, and transaction signing. Generation of the (epk, esk) keypair and signing of transactions are done by compiled Rust binaries.

The flow of the service is as follows:

Algorithm 2 — User client flow.

1. User clicks sign in button, site generations a (epk, esk) keypair and the blinding key r.
2. The site calculates the nonce and redirects to Google OAuth.
3. Upon authorizing, OAuth redirects back to the site which generates a random u, hashes the user email E, computes $h = g^u E$ to send to the pepper service through a redirect.
4. The site receives the encrypted pepper through a redirect and decrypts it using the pepper service's public key.
5. The site displays relevant JWT, pepper, and key information on a homepage with an option to create a transaction with destination and quantity fields.
6. Creating a transaction signs it with the users (epk, esk) pair and displays the signed transaction data.

§4.2 Pepper Service

The pepper service is written purely in Python and Flask and can be found in [3]. It simply receives h from a redirect, exponentiates it by the pepper service's secret key, and redirects back to the user client.

§4.3 Prover

Our prover is run locally using OpenVM and Rust and can be found in [2]. It takes in the JWT, nonce, pepper p, and blinding key r as hidden inputs; technically, the nonce is not needed and can be computed by the prover with $H(epk, r)$ and epk as a public input. The prover verifies the JWT's signature and that the nonce matches $H(epk, r)$ and generates a zkVM proof of computation. It also reveals the expected address of the given inputs.

§5 Potential Future Improvements

Due to time restraints, we were unable to finish our implementation completely. Some future avenues we would like to explore:

- Our zero-knowledge proving service is disjoint from our main protocol. This was mainly because it would be incredibly computationally intensive to compile our prover to use the recursive Halo2 proof required to deploy to the EVM, but we could still implement this in the future.
- For the same reason as above, we were unable to deploy our protocol onto Ethereum contracts, but it would be interesting to do so in the future.
- The security of the pepper service could be improved by increasing the number of bits of the pepper keys and upgrading to an elliptic curve discrete log system.
- Our zero-knowledge protocol is only computationally binding. Although we believe that it is not computationally feasible to reverse engineer user information from this information, we could take more effort to verify this claim, and take steps to make the protocol statistically hiding.

§6 Contribution

- Alex and Kshitij worked on implementing the zero knowledge proofs using OpenVM.
- Thomas and Tony worked on implementing the main protocol.

References

[1] https://alinush.github.io/keyless

[2] https://github.com/cocohearts/aptos-clone

[3] https://github.com/oursaco/65610