Zero-Knowledge Proofs for browser-based private authentication to blockchain

Peeter Laud

With Martin Pettai, Andres Ojamaa, Martin Suomalainen

Using existing credentials and MPC-in-the-Head for browser-based private authentication to blockchain

Andres Ojamaa
Martin Pettai
Martin Suomalainen
Peeter Laud

: Cybernetica AS

10 May 2026

ZKPROOF

Paving the path to adoption

8th Workshop

Rome, Italy
9-10 May 2026

Authentication in blockchain

Problem

- User wants to post a transaction to the blockchain
  - Authenticated, i.e. signed
- User does not want to manage the private keys

Solution

- Get a JWT from some OAuth provider (Google, Apple, …)
  - Authenticates the user
  - Contains the transaction
- Post the JWT to blockchain

New problem

JWT payload has sensitive fields (name, e-mail, photo, etc)

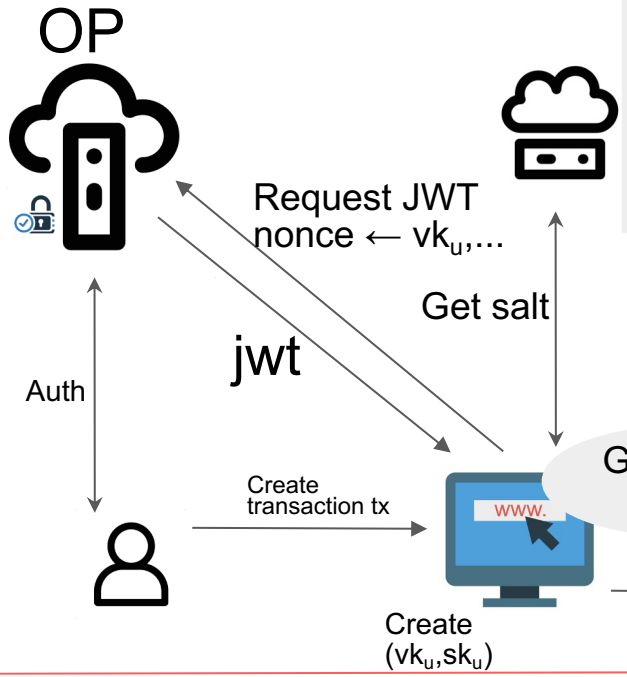
New solution

zkLogin: Privacy-Preserving Blockchain Authentication with Existing Credentials

Foteini Baldimtsi, Mysten Labs, George Mason University
Konstantinos Kryptos Chalkias, Mysten Labs
Yan Ji, Cornell University
Jonas Lindstrøm, Mysten Labs
Deepak Maram, Mysten Labs
Ben Riva, Mysten Labs
Arnab Roy, Mysten Labs
Mahdi Sedaghat, imec-COSIC, KU Leuven
Joy Wang, Mysten Labs

@ CCS 2024

zkLogin flow



$$\text{Ckt}\left(\begin{matrix}\text{zkx} = (\text{pk}_{OP}, \text{iss}, \text{zkaddr}, T, vk_u),\\ \text{zkw} = (\text{jwt}, \text{salt}, r)\end{matrix}\right):$$
$$\text{zkaddr} = H(\text{jwt.stid}, \text{jwt.aud}, \text{jwt.iss}, \text{salt}) \text{ and } \text{jwt.iss} = \text{iss} \text{ and}$$
$$\text{jwt.nonce} = H(vk_u, T, r) \text{ and } JWT.\text{Verify}(\text{pk}_{OP}, \text{jwt}).$$

Parse jwt

- RSA or ECDSA verify
- SHA256

Problem

- Proof generation is expensive
- zkLogin outsources the generation

Generate
zkp

Groth16

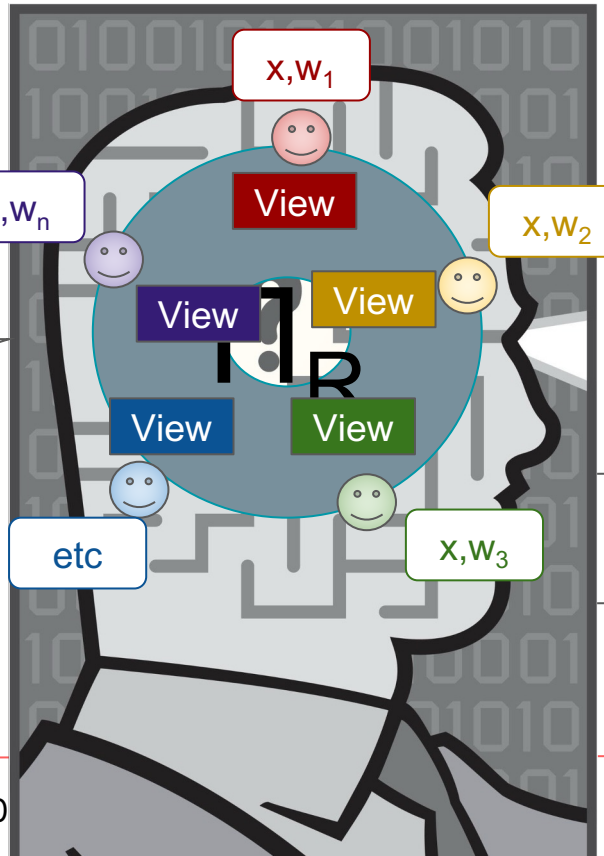
Post tx, vk$_u$, T, sig(sk$_u$, tx), zkp

Motivation + “Research question”

- Alphabill blockchain: https://github.com/alphabill-org/alphabill
  - Sharded ledger with heterogeneous rules; fast consensus with “root→shard” trust in validation; programmability via WASM
  - Was developed by Guardtime
- Can the proof be generated in browser?
  - By using some other NIZK technique, perhaps non-succinct?
- CHESS: Cyber Security Excellence Hub in Estonia and South Moravia
  - Partners include Cybernetica, Guardtime, [many more]

CHESS

MPC in the Head (MPCitH)



Witness w ∈ **W**

Secret Share

$w_1, w_2, \ldots, w_n$

x,w_n

Relation R, where R : **X** × **W** → {0,1}

Instance x ∈ **X**

Secure against **two passive** corruptions

$\Pi_R(x; w_1,\ldots,w_n) = R(x,w)$

Com(View), Com(View), …, Com(View)

■, ■

View, View

FS

- Output = 1?
- Consistent?

Soundness error < $1 - 1/n^2$

CYBERNETICA

Why would MPCitH be suitable?

- Computations in “zkLogin relation” are heterogeneous:
  - Signature verification: large fields
  - Hash function (in signature verification): bits
    - More freedom to choose some other hash functions
  - Parsing: characters, i.e. 8-bit values
  - Base64 encoding: 6- and 8-bit values
- Passively secure MPC can match that heterogeneity

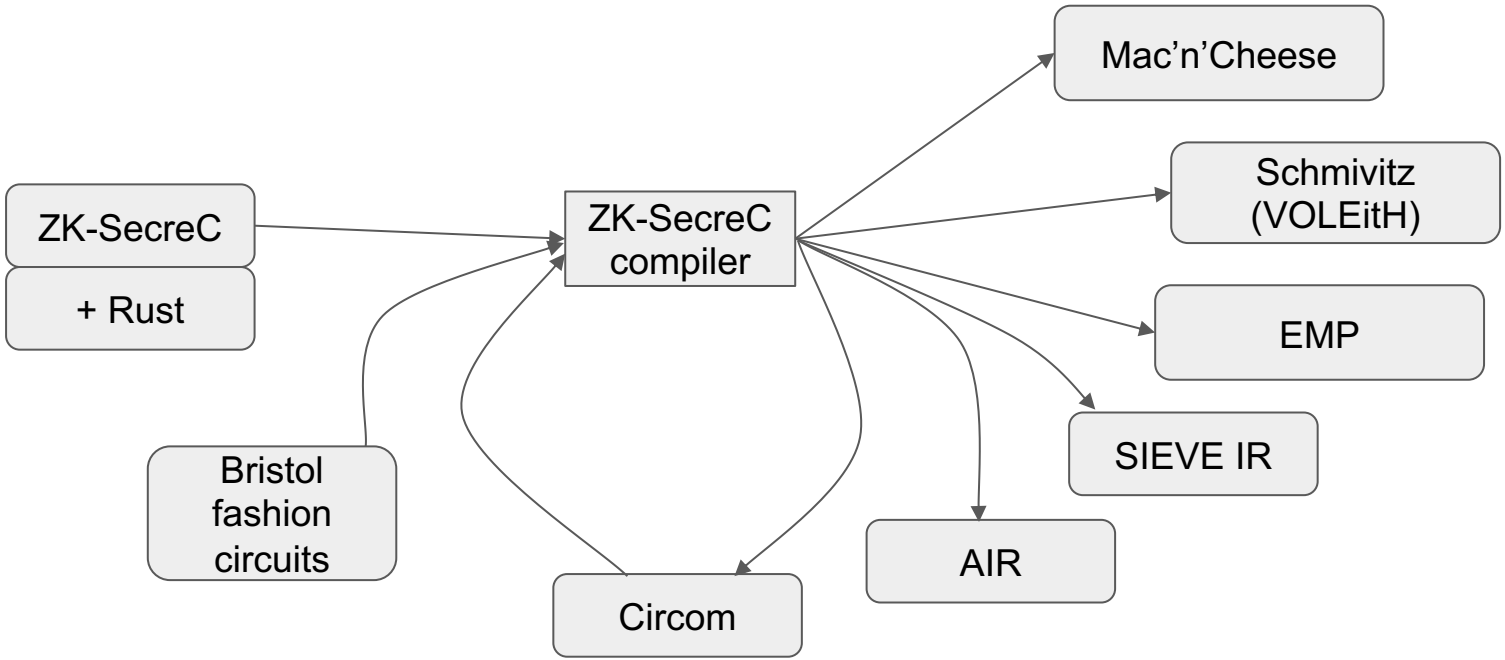
MPCitH implementations in Github

- ZKB++
  - Very simple protocol
  - A bit of documentation
  - Can read the paper and the code in parallel
    - Consisted of ca. 6k LoC of C++
- Katz-Kolesnikov-Wang
  - No documentation. Could not get it to run

ZK-SecreC

- DSL for statements one may want to prove in zero-knowledge
- Information-flow type system, with two dimensions:
  - Is a value known only to prover, or also to verifier? Or is it a public constant?
  - Is a value held locally? Or {{managed by the ZKP technique}}?
- Imperative language. Some useful control-flow constructions
- A variety of data types (for “integers”)

ZK-SecreC toolchain



Support for many rings in ZKB++

- Supported so far: $\mathbb{Z}_M$, M is a k-bit prime, where $k \in \{3,...,272\}$
- Also supported so far: $GF(2^k)$, where $k \in \{3, 17, 33, 65\}$

Our additions

- $\mathbb{Z}_p$ for p the size of P-256 & P-384 prime fields, and EC groups
- $\mathbb{Z}_M$ for $M \in \{2^{32}, 2^{64}\}$
- $\mathbb{Z}_{2^k}$ for $k \in \{1, \ldots, 64, 256, 384\}$ (specialized $\mathbb{Z}_{2^{32}}$ and $\mathbb{Z}_{2^{64}}$)
- Elliptic curve *groups* (specialized P-256 & P-384)
- The use of several rings at the same time
  - MPCitH protocol to convert between rings

Support for rings in ZK-SecreC type system

n :: Nat
∞ :: Nat

uint :: Nat → Unqualified
bool :: Nat → Unqualified

- uint[n] is $\mathbb{Z}_n$. uint[∞] is $\mathbb{Z}$
- bool[n] is booleans, but compatible with uint[n]

The kind for “pure” data types

bitwise :: Nat → Ring
plain :: Nat → Ring
int :: Ring → Unqualified
bin :: Ring → Unqualified

- uint[n] ≡ int[plain[n]]
- bool[n] ≡ bin[plain[n]]

- ZK-SecreC uses SIEVE IR’s “Circuit Configuration Communication” syntax
  - Compiler learns capabilities of the ZKP back-end
    - rings, conversions, plugins…
- This has been extended

Elliptic curve groups: “undocumented API calls”

Interesting parts of the relation

$$\mathrm{Ckt}\left(\begin{matrix}\mathrm{zkx} = (\mathrm{pk}_{OP}, \mathrm{iss}, \mathrm{zkaddr}, T, vk_u),\\ \mathrm{zkw} = (\mathrm{jwt}, \mathrm{salt}, r)\end{matrix}\right):$$
$$\mathrm{zkaddr} = H(\mathrm{jwt.stid}, \mathrm{jwt.aud}, \mathrm{jwt.iss}, \mathrm{salt}) \text{ and } \mathrm{jwt.iss} = \mathrm{iss} \text{ and}$$
$$\mathrm{jwt.nonce} = H(vk_u, T, r) \text{ and } JWT.\mathrm{Verify}(\mathrm{pk}_{OP}, \mathrm{jwt}).$$

- Parsing JWT (correct JSON only, coming from OP)
  - Slicing an array, running checks on that slice
- Base64 encoding / decoding
- SHA-256 computation
- ECDSA verification (no RSA)

Our contributions to implementing these parts

Array slicing

- Recursive approach to pointing out the location of the slice
- Asymptotically decreases the size of the relation

Base64

- More efficient formula for conversion
- Bit-parallel execution

ECDSA verification

- Nothing in particular…
  - Leverage the implementation of various rings for ZKB++
  - Sharing over EC group reduces proof size

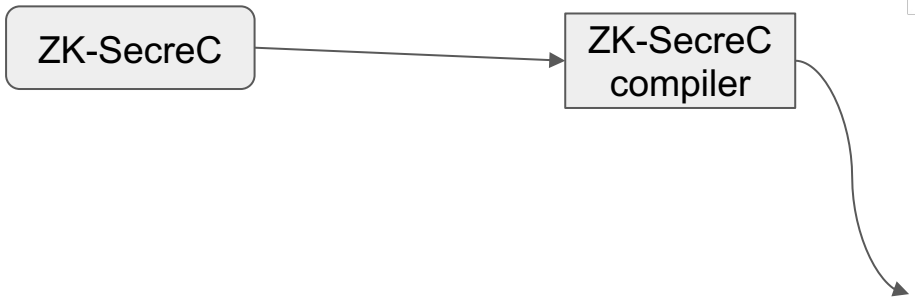
SHA-256

- Bit-parallel execution of the rounds of the compression function
- Bit-parallel execution of block finishes
  - Careful scheduling, supported by ZK-SecreC

More additions to ZKB++

- Multiple instance and witness values
- Multiple output values
  - Π_R for the relation R “accepts” if all outputs are 0
- Declassification and random number generation in the MPCitH protocol
- Serialization and deserialization of the proof

ZK-SecreC — ZKB++ integration



ZKB++ (and our additions)

```
CircuitContainer* c = new CircuitContainer();
```

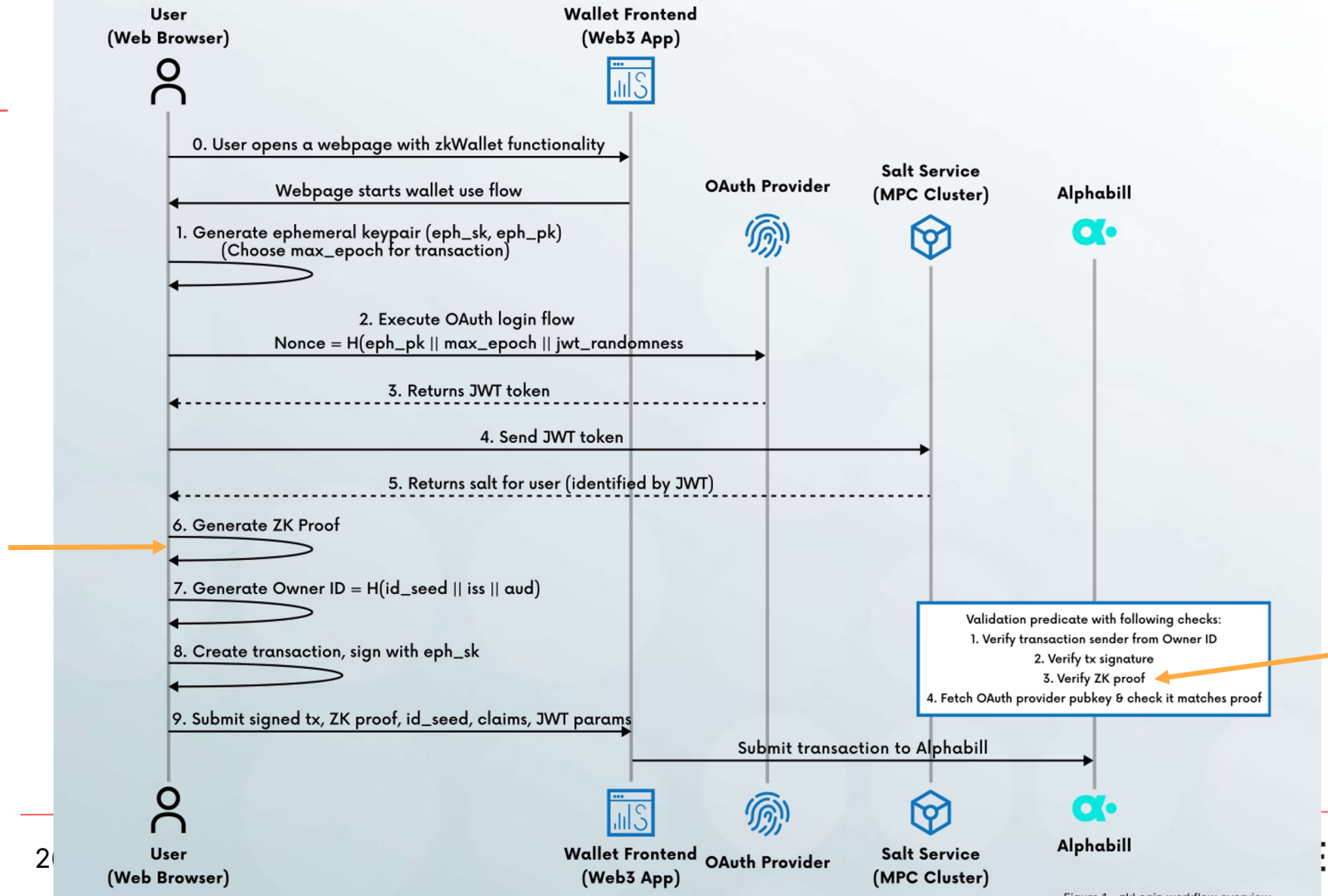
```
CircuitContainer::CircuitContainer() {

}

CircuitContainer::~CircuitContainer() {
  if(this->value_shares_ != NULL) {
    for(uint32 i = 0; i < this->party_size_; i++) {
      delete[] this->value_shares_[i];
```

Benchmarks (local execution)

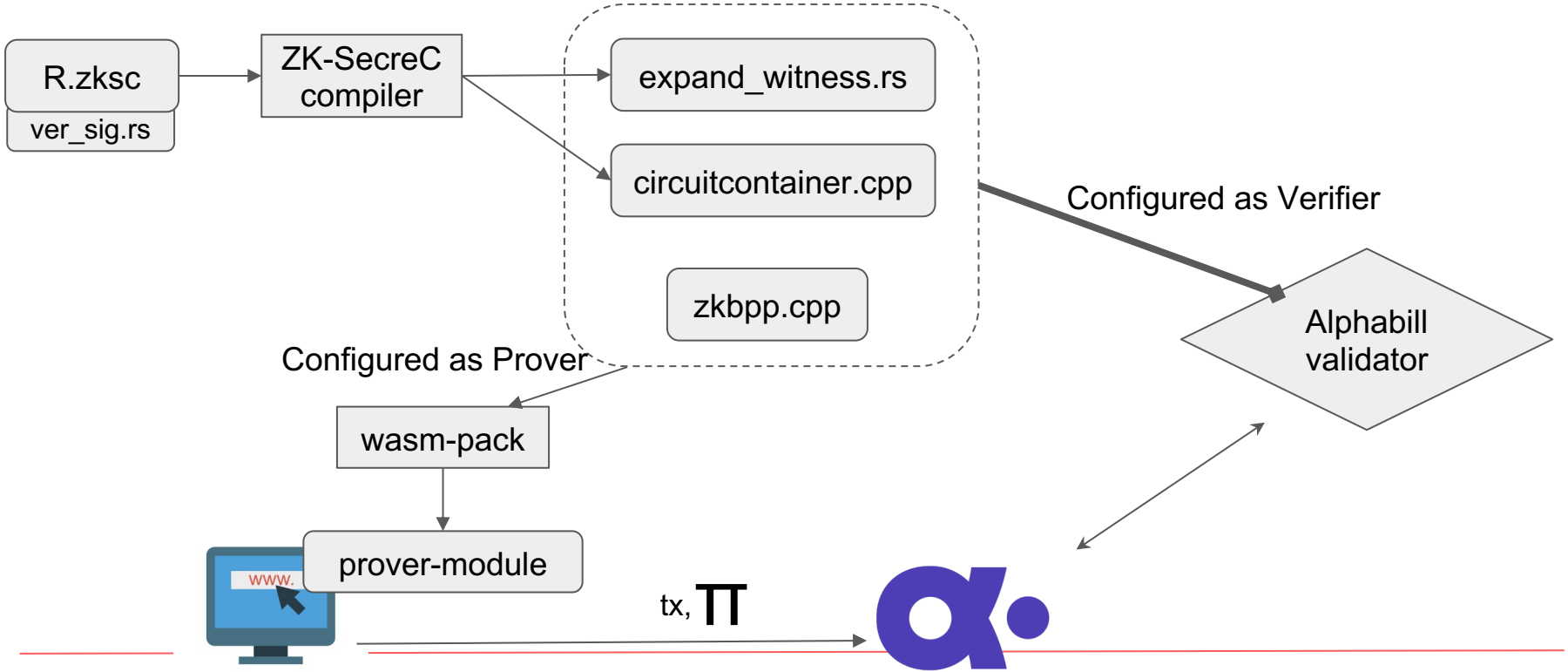
	prover	verifier	witn. expand	size
init + parse + check + base64	0.45s	0.32s	0.09s	15.8 MB
… + SHA	1.49s	1.09s	0.12s	45.5 MB
Full circuit with 256-bit ECDSA in ZKSC	2.06s	1.48s	0.18s	66.2 MB
Full circuit with 256-bit ECDSA in ZKB++	2.08s	1.5s	0.33s	49.1 MB
Full circuit with 384-bit ECDSA in ZKSC	2.54s	1.77s	0.25s	90.1 MB
Full circuit with 384-bit ECDSA in ZKB++	3.01s	2.11s	0.6s	50.8 MB

Execution on a 2-year old laptop

1472-byte payload. Fixed-length payloads and header. Variable-length adds ca 10%


User
(Web Browser)
Wallet Frontend
(Web3 App)
OAuth Provider
Salt Service
(MPC Cluster)
Alphabill
0. User opens a webpage with zkWallet functionality
Webpage starts wallet use flow
1. Generate ephemeral keypair (eph_sk, eph_pk)
(Choose max_epoch for transaction)
2. Execute OAuth login flow
Nonce = H(eph_pk || max_epoch || jwt_randomness
3. Returns JWT token
4. Send JWT token
5. Returns salt for user (identified by JWT)
6. Generate ZK Proof
7. Generate Owner ID = H(id_seed || iss || aud)
8. Create transaction, sign with eph_sk
9. Submit signed tx, ZK proof, id_seed, claims, JWT params
Validation predicate with following checks:
1. Verify transaction sender from Owner ID
2. Verify tx signature
3. Verify ZK proof
4. Fetch OAuth provider pubkey & check it matches proof
Submit transaction to Alphabill
User
(Web Browser)
Wallet Frontend
(Web3 App)
OAuth Provider
Salt Service
(MPC Cluster)
Alphabill

Running in browser + blockchain



Benchmarks (browser)

Full circuit with 256-bit ECDSA in ZKB++

	witn. exp.	Prover
Firefox 150.0.1 (64bit)	0.4s	4.5s
Chrome 147.0.7727.137 (64-bit)	0.4s	4.3s

CPU: AMD Ryzen 7 PRO 8840U
OS: NixOS, kernel 7.0.3

Thank you

https://cyber.ee/

info@cyber.ee

cybernetica

CyberneticaAS

cybernetica_ee

Cybernetica