Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

Сервіс анонімізації транзакцій в блокчейні Substrate шляхом криптографічного шифрування входів і виходів мікшера

Текстова частина до кваліфікаційної роботи

за спеціальністю «Комп'ютерні науки» - 122

Керівник кваліфікаційної роботи

Старший викладач

Гороховський К.С.

___________________ (Підпис)

“___” _____________ 2023 року

Виконав студент

КН-4 Михайленко О.І.

“___” _____________ 2023 року

Київ 2023

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Старший викладач

__________ Гороховський К.С.

„_____” _____________ 2022 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на кваліфікаційну роботу

студенту Михайленку Олександру Ігоровичу

факультету інформатики 4 курсу бакалаврської програми

ТЕМА: **Сервіс анонімізації транзакцій в блокчейні Substrate шляхом криптографічного шифрування входів і виходів мікшера**

Зміст ТЧ до кваліфікаційної роботи:

Дата видачі „___” _________ 2022 р.

Керівник_______________

(підпис)

Завдання отримав_______________

(підпис)

Календарний план виконання роботи

№	**Назва етапу кваліфікаційної роботи**	**Термін виконання етапу**	**Примітка**
1.	Отримання завдання на кваліфікаційну роботу	01.10.2022	
2.	Огляд літератури за темою роботи	01.11.2022	
3.	Проведення дослідження	01.12.2022	
4.	Написання програмного застосунку	01.01.2023	
5.	Написання текстової частини	04.04.2023	
6.	Захист кваліфікаційної роботи	30.05.2023	

Студент ______________________________

Керівник _____________________________ “_____”_______ 2022

Зміст

Анотація

Робота присвячується дослідженню використання доказів з нульовим розголошенням у блокчейн застосунках. Для розв’язання проблеми приватності переказів криптовалюти користувачами використана бібліотека для генерації та верифікації доказів з нульовим розголошенням Plonk, мова Rust та domain-specific language “ink!”, а також мова Typescript для написання сервера реле.

Розроблено новий смарт контракт, який може розгортатися на блокчейн-мережах, побудованих за допомогою фреймворку Substrate. Також розроблена бібліотека “plonk prover”, в якій імплементована схема для генерації доказів з нульовим розголошенням та допоміжні функції для роботи з ними, розроблені структури даних, утиліти для компіляції Rust коду в JavaScript код, сервер реле для приватного відправлення транзакцій виводу коштів зі смарт контракту, і утиліта командного рядка для роботи з бібліотекою “plonk prover”.

Текстова частина кваліфікаційної роботи містить опис дослідження можливості використання Plonk у WebAssembly смарт контрактах та імплементацію криптографічного мікшера коштів. Текстова частина кваліфікаційної роботи також описує необхідні теоретичні відомості про використані інструменти та технології.

Ключові слова: блокчейн, смарт контракт, Plonk, доказ з нульовим розголошенням, WebAssembly, Substrate, Rust, Typescript

A zero-knowledge proof-based token mixer in Substrate

Keywords: blockchain, smart contract, Plonk, zero-knowledge proof, WebAssembly, Substrate, Rust, Typescript

Abstract

In this work, we explore the possibility of implementing a cryptocurrency mixer specifically in the form of a smart contract that can be deployed to any *Substrate*-compatible chain.

Zero-knowledge-proof technology has become increasingly popular in anonymity services. They possess the power to hide or obfuscate secret information while still being able to provide means of public verification of this information. Zero-knowledge proofs are fit for the use case described here - we will be able to detach the monetary transaction origin from the transaction target, circumventing the transparency by which most public general-purpose blockchains function.

Please note that a cryptocurrency mixer is not a new idea. The project we drew inspiration from is Tornado.Cash. Consequently, the logic of constructing the proofs will be similar to the zk-SNARK circuits of Tornado.Cash. However, since blockchain technology is abundant, as evidenced by the existence of many blockchains, such as Ethereum, Substrate, Solana, NEAR, and many others, developers need to create potentially different implementations of similar logic in smart contract formats that are specific to those blockchains. Substrate specifically had unfinished implementations of cryptocurrency mixers but in the form of *Substrate pallets*, which are different from smart contracts.

Our contribution

In this work, we explored how to implement a cryptocurrency mixer for use in Substrate-compatible chains like the parachains of Polkadot and Kusama. It is different from the original blockchain used to support Tornado.Cash [1], Ethereum, in that it uses WebAssembly as the binary format for smart contracts and a custom-developed WebAssembly execution environment for smart contract execution. Mainly, we used *ink!* as the smart contract language of choice to produce WebAssembly code and smart contract application binary interface, Plonk [2] as the ZK-SNARK of choice, and Poseidon [3] as the family of hash functions. To add to that, we unify client-side and blockchain-side operations to produce and verify proofs in one library that supports cross-compiling into different targets. We successfully built an *ink!*-based cryptocurrency mixer called $Slushie$.

The technical idea

Blockchain technology allowed users to transfer cryptocurrency and interact with programs in a decentralized manner. Since blockchains typically use user public keys or some derivatives of them, like hashes of the said private keys, some tend to think that it makes the technology anonymous. However, it requires significant efforts to achieve anonymity in blockchains due to the abundance of communication points.

Let's use Ethereum for our next example. Suppose an address $\mathbf{A}$ would like to send $\mathbf{X}$ ETH to address $\mathbf{B}$. As Dr. Gavin Wood explains [4, p.4], the transaction, among other fields, contains the $\mathbf{to}$ field, which is described as

“...the 160-bit address of the message call’s recipient…”, and fields $\mathbf{r}$, $\mathbf{s}$, which are “...used to determine the sender of the transaction…”. To paraphrase, we can simplify the aforementioned definitions and say that *each transaction in Ethereum contains sufficient information about the sender and the receiver of said transaction*. In this specific definition, by *sufficient* we mean that we know how to identify both the sender and the receiver of the transaction. We can recover the public key of the sender from the signature of the transaction: $publicKey = ECDSARECOVER(e, r, s)$ [4, p.26].

Public keys themselves do not tell any information about the user. However, providers of public RPC endpoints, which accept user transaction requests, can connect the IP address of the sender of the transaction to their public key. Even though they do not know the public key of the destination, once the account with the destination address is used, its public key can be recovered, too.

Considering the above, we can say that the *privacy of the sending and the receiving party is limited to the networking layer*. This makes public blockchains like Ethereum pseudo-anonymous – RPC endpoint providers know about the relationship between the sender’s public key and their IP address.

Tornado.Cash [1] developed a solution that greatly increases the privacy of the users by decoupling the sender origin from the receiver by mixing the funds being sent in a smart contract.

Let's define what the term "mixer" means. In this work, the term mixer means a smart contract that defines two state-mutable messages, a $deposit$ message, and a $withdraw$ message, and uses zero-knowledge proofs to convince the smart contract about the authenticity of the claim to withdraw funds for a specific deposit. The $deposit$ transaction will be responsible for transferring some predefined token value to the smart contract account, while the $withdraw$ transaction will accept some public information about the deposit, verify that information, and transfer the same predefined token value to the caller account.

The above was implemented by [1] in [5]. In the scope of this work, we implement the same idea expressed by Pertsev et. al. using a different blockchain technology, *Substrate*, and make improvements to the project by reusing exactly the same code for producing and verifying proofs across the on-chain side and the off-chain side.

Substrate

Substrate is a framework for building Layer 1 blockchains, which provides the developers with pluggable modules, *pallets*, and various consensus algorithms, which include both block production and finalization algorithms and more customizable components. Blockchains built with this framework are highly configurable w.r.t block production time, block finalization time, block length, maximum block weight (an analog for *gas* in other blockchains), and weight usage per second.

There are some important decisions as to why we use *Substrate*:

1. *Substrate* implements its smart contract functionality in a pluggable module, called *pallet-contracts*. It allows any blockchain built with Substrate to include this module at genesis or at runtime *without hard forks*.
2. *pallet-contracts* is a *Substrate* pallet, which defines a set of callable transactions that manage the execution of smart contract calls, uploading of the smart contract WebAssembly code, instantiation of the code, and removal of the code. The smart contracts deployed with *pallet-contracts* must conform to its *metadata* format, which is an analog of ABI in other blockchains. The WebAssembly format is not specified, though, enabling developers to use languages like *ink!* and *AssemblyScript* to compile to *pallet-contracts*-conformat format.

3. Since zero-knowledge proof verification is a computation-intensive operation, we make use of the fact that *Substrate* allows almost arbitrarily changing the maximum block production time. Instead of processing transactions that would outweigh any block on a production *Substrate parachain*, we drastically increase the maximum block weight to ensure the $withdraw$ transaction will not outweigh the block.
4. *ink!* is an eDSL (embedded DSL) that is highly interoperable with *Substrate*. We use *ink!* to build $Slushie$ because *ink!* is Rust-based, statically typed, has extensive documentation, and supports generating WebAssembly code and *pallet-contracts*-conformant metadata.

Plonk

Plonk [2] is a universal fully-succinct zk-SNARK, by which we shall mean a system that allows one to construct zero-knowledge proofs and verify them. We choose Plonk over other zero-knowledge proof systems because it has implementations [6] in Rust that can compile to WebAssembly bytecode. In using WebAssembly as the compilation target of choice, we achieve two things: the ability to reuse the code in client-side environments (by using JavaScript wrappers over WebAssembly code) and the ability to use the proof verification logic in the *ink!* smart contract.

Slushie

$Slushie$ is a smart contract that allows users to carry out private transfers. It draws heavy inspiration from Tornado.Cash, but it does have some notable differences.

First, we use a different blockchain (Substrate-based chain instead of EVM-based). Second, instead of checking a pairing as the verification process, we directly import the verification function for our circuit that uses Plonk. We can do that since both Plonk and Slushie can compile to WASM. Generally speaking, to construct proofs with Plonk for any quadratic arithmetic program, we need to go through 2 steps: generate an SRS (structured reference string) and model the circuit by writing appropriate code in Rust. While we recommend always using a trusted setup ceremony for initializing an SRS, we omit this step in the scope of our work.

$Slushie$ is a Rust-based project which consists of four core crates: the relayer server, the prover, the prover CLI, and the smart contract. Other two additional components are Typescript-based integration tests and a Rust crate with shared type definitions and utility functions. Let us go through the components one by one and explain the role of each one.

The relayer server is a Typescript application that starts an HTTP server with one endpoint that allows submitting $withdraw$ transactions without revealing the sender's identity. It serves as the off-chain communication layer between the end user and the smart contract, creating blockchain transactions from the single relayer account with the specified parameters, supplied in the POST request body. It decouples the sender's IP address from the blockchain transaction (the relayer does not collect any information about users' IP addresses in our implementation) by delegating the transaction signing to some static public key pair, which is securely held within the relayer. It cannot steal the funds of the users as the recipient is supplied as a public input to the proof verification procedure, meaning that if the relayer supplies some different recipient address instead of the address that was originally used to construct the proof, the transaction will fail.

The prover is a Rust crate that defines the proof construction logic (the circuit), the proof verification logic, commitment generation utilities, prover & verifier data generation utilities, Merkle tree implementation, hasher-related code (our

Merkle trees can use different hashers), and JavaScript bindings for relevant functions.

The plonk prover CLI (plonk-prover-tool) is a wrapper over the prover, which allows us to conveniently generate relevant data (commitments, prover/verifier data, opening keys, etc.) by invoking CLI subcommands.

$Slushie$ is an *ink!*-based smart contract deployed to pallet-contracts-compatible networks. It also holds the other Merkle tree implementation, which we'll talk about later in detail.

Slushie (smart contract)

$Slushie$ is a smart contract built with *ink!* eDSL. It has two callable state-mutable messages, $deposit$ and $withdraw$, and one query, get_root_hash.

The $deposit$ method accepts the user's generated commitment as the only input. Essentially, depositing funds to $Slushie$ happens both on the off-chain user side (front-end or CLI) and the on-chain side (smart contract transactions). On the off-chain side, the user generates their commitment by randomly selecting two numbers, the nullifier k and the randomness r, and hashing them together using a ZK-friendly hash (in our case, Poseidon): $h = Poseidon252(k||r)$. On the on-chain side, the user calls the smart contract transaction with the data generated on the off-chain side.

Generally, smart contracts consist of storage and callable messages. On top of having storage and callable transactions, $Slushie$ also defines the contract error type and the events, which we use to communicate notable state changes in the contract. For example, one deposit call completely changes the underlying Merkle tree in the storage, motivating us to communicate this state change via an event. The contract error enumeration is merely a collection of all errors that can happen in the contract's transactions.

Plonk prover

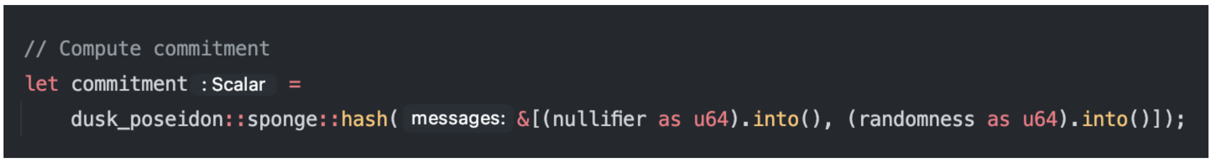
Plonk prover consists of the following logical parts: proof generation, proof verification, zk-SNARK circuit definition, commitment generation (not to be mistaken with commitments in ZKP context), the flat Merkle tree implementation, the hashers, javascript bindings, the Merkle implementation with root history, public parameters setup utilities, and general utilities.
Arguably, the two most important parts of the crate are the Merkle trees and the circuit. The reason for having two different implementations of the Merkle tree is that the flat Merkle tree is used for gathering the openings, meaning the values of the sister nodes on the way from leaf l to the root R, while the Merkle tree with history makes sure that we can verify the proof by checking that the root R is valid, and thus the Merkle tree can be recreated. The circuit, on the other hand, describes the logic by which we generate and verify the proofs. Similarly to [1], our quadratic arithmetic program [7, p.42] consists of two general parts: verifying that the nullifier hash h, supplied as a public input, is equal to the result of hashing the nullifier k, supplied as a private input, and that the root R, supplied as a public input, is equal to the root computed by reconstructing the Merkle tree from the commitment c, the path p and the openings o and taking its root.

Commitments

A commitment in the context of this work is a 32-byte result of applying hash H to the nullifier k and the randomness r (both are non-negative 32-bit integers). The commitments are used to uniquely identify deposits in the system.

In order to make a deposit to $Slushie$, the user of the system must first generate the commitment using a cryptographically secure pseudorandom number generator (abbreviated as $CSPRNG$). In the course of this work, we will assume that the native operation system randomness source is a $CSPRNG$.

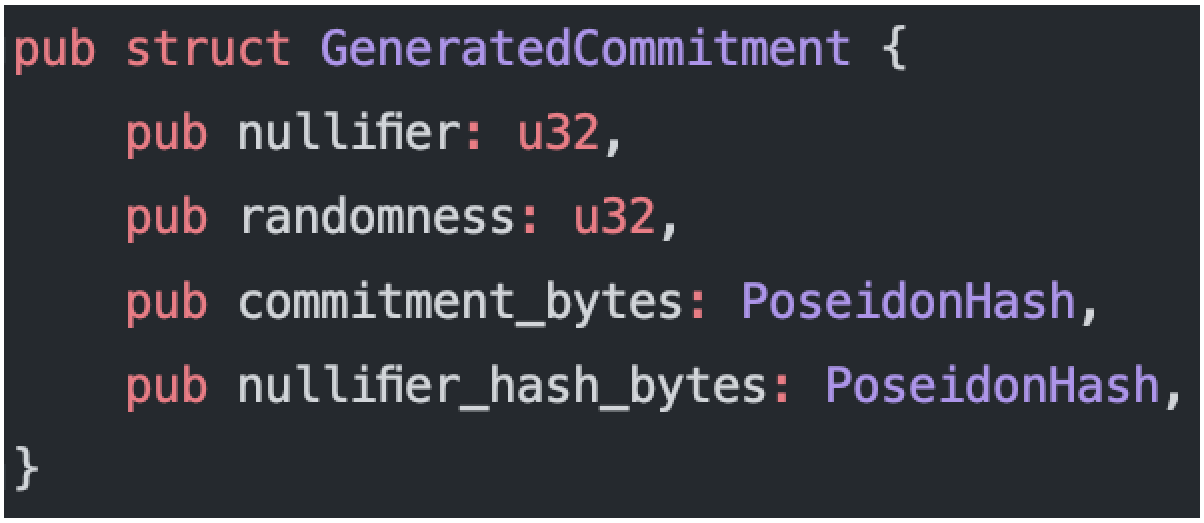
As for the implementation, the commitment generation is handled by the *dusk_poseidon* [6] crate.



```
// Compute commitment
let commitment :Scalar =
    dusk_poseidon::sponge::hash( messages: &[(nullifier as u64).into(), (randomness as u64).into()]);
```

Fig. 1, commitment generation

As a convenience, we also provide the generated nullifier, and nullifier hash together with the commitment, since all those values must be used in order to both generate a proof and provide public inputs for the contract to verify the supplied proof.



```
pub struct GeneratedCommitment {
    pub nullifier: u32,
    pub randomness: u32,
    pub commitment_bytes: PoseidonHash,
    pub nullifier_hash_bytes: PoseidonHash,
}
```

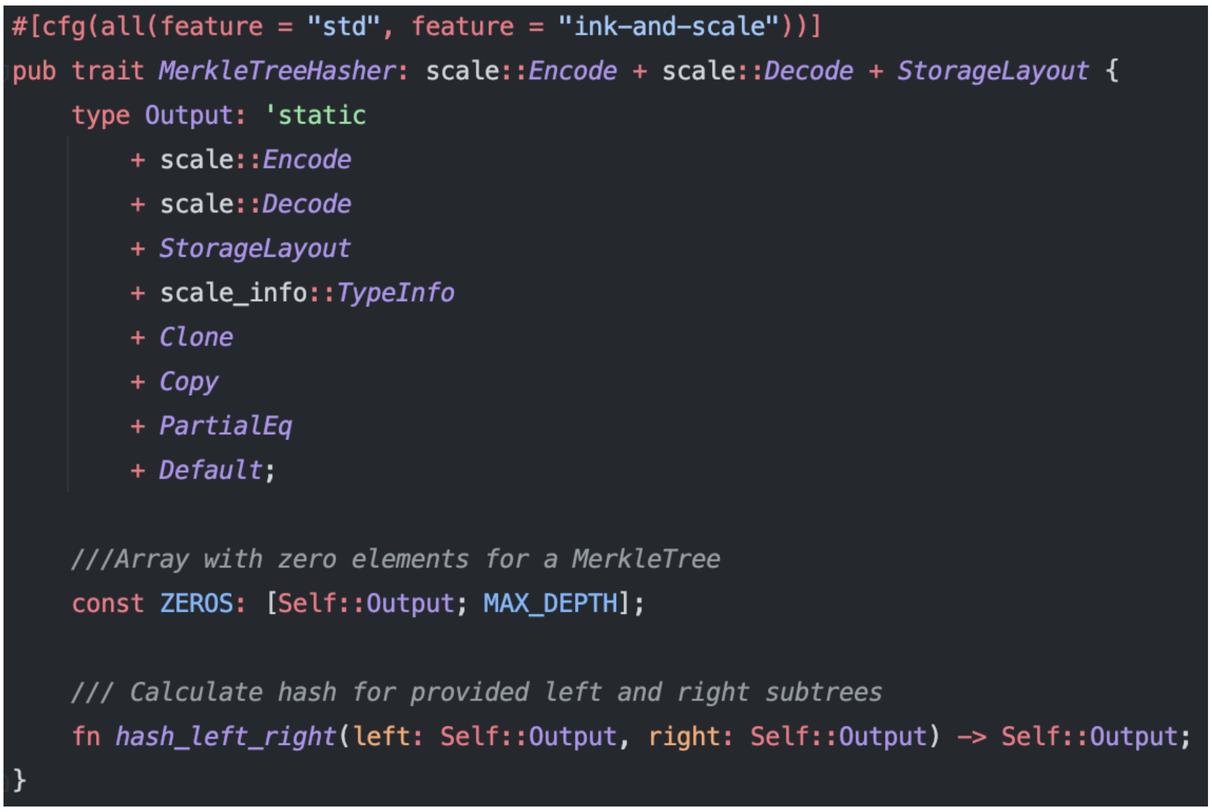
Fig. 2, generated commitment structure

Hashers

Slushie aims to be hasher-agnostic, meaning that we can use different hashers with the contract and the proof system. For this, we have created a Rust trait, called $MerkleTreeHasher$, which is the interface for all hashers that our Merkle tree can use.

In our implementation, we defined three copies of the same trait with slightly different trait bounds on the $Output$ associated type due to the requirement for our crates to compile to the WebAssembly binary format. Since the hashers are structs that are stored in the contract storage, they need to implement some traits that are only available in *std* compilation mode which cancels our ability to compile the code to WebAssembly.

The trait definition that we use in our contract for generating contract metadata (ABI):



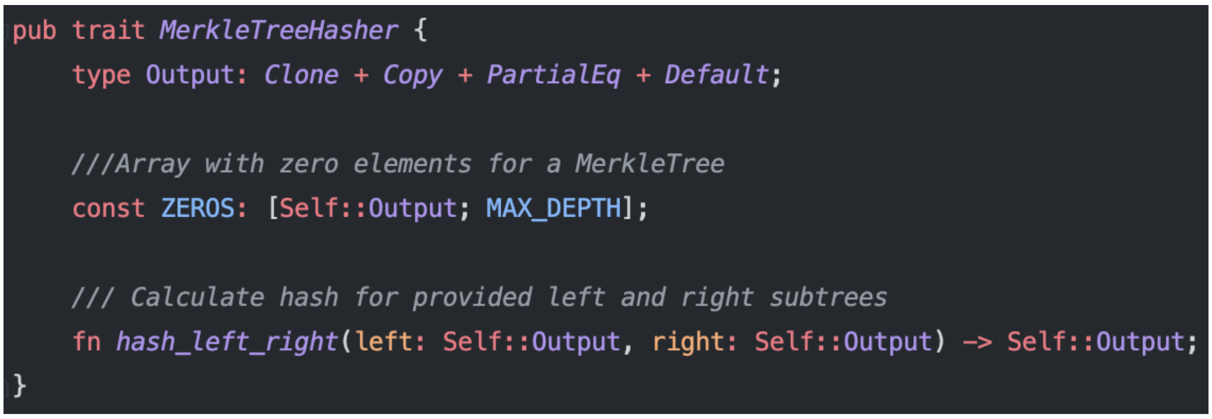
```
#[cfg(all(feature = "std", feature = "ink-and-scale"))]
pub trait MerkleTreeHasher: scale::Encode + scale::Decode + StorageLayout {
    type Output: 'static
        + scale::Encode
        + scale::Decode
        + StorageLayout
        + scale_info::TypeInfo
        + Clone
        + Copy
        + PartialEq
        + Default;

    ///Array with zero elements for a MerkleTree
    const ZEROS: [Self::Output; MAX_DEPTH];

    /// Calculate hash for provided left and right subtrees
    fn hash_left_right(left: Self::Output, right: Self::Output) -> Self::Output;
}
```

Fig. 3.1, *std* version of MerkleTreeHasher

The trait definition that we use in WebAssembly but not in the contract:



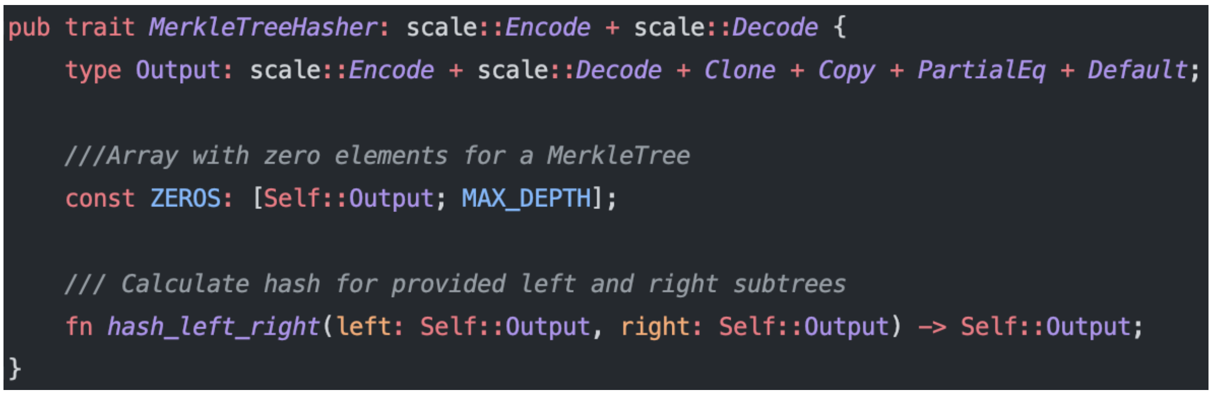
```rust
pub trait MerkleTreeHasher {
    type Output: Clone + Copy + PartialEq + Default;

    ///Array with zero elements for a MerkleTree
    const ZEROS: [Self::Output; MAX_DEPTH];

    /// Calculate hash for provided left and right subtrees
    fn hash_left_right(left: Self::Output, right: Self::Output) -> Self::Output;
}
```

Fig. 3.2, *no_std* version of MerkleTreeHasher

The trait definition that we use in our contract for compiling the contract to WebAssembly:



```rust
pub trait MerkleTreeHasher: scale::Encode + scale::Decode {
    type Output: scale::Encode + scale::Decode + Clone + Copy + PartialEq + Default;

    ///Array with zero elements for a MerkleTree
    const ZEROS: [Self::Output; MAX_DEPTH];

    /// Calculate hash for provided left and right subtrees
    fn hash_left_right(left: Self::Output, right: Self::Output) -> Self::Output;
}
```

Fig. 3.3, *no_std* version of MerkleTreeHasher with SCALE encoding support

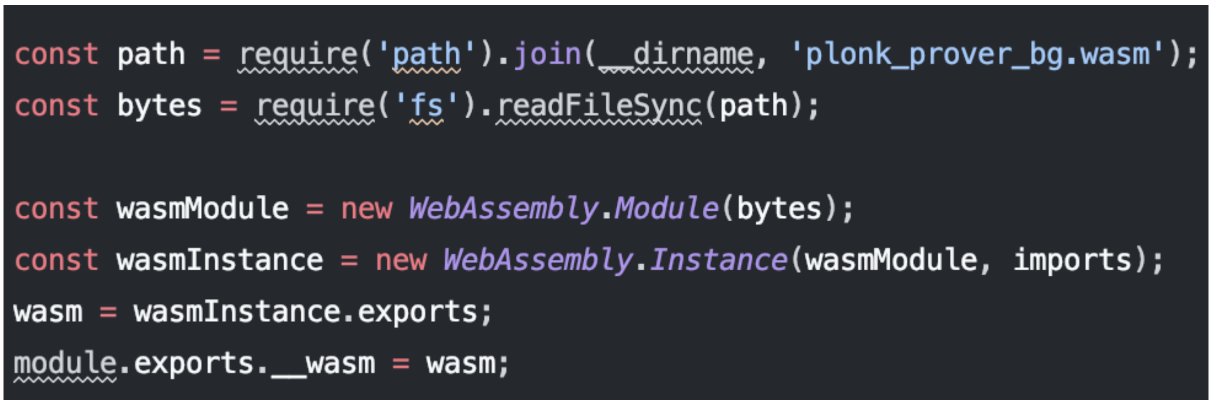
As for the trait implementations, currently *Slushie* supports two hashers: a Poseidon-based hasher and a Blake2-based hasher.

JavaScript bindings

Since *plonk_prover* can be directly compiled to WebAssembly, it is feasible to reuse the same functions used by *plonk_prover_cli* in JavaScript packages that import WebAssembly modules. This approach produces less code and increases the overall implementation security of the project.

These JavaScript packages can be later used on the front-end side to generate and verify proofs.

To generate JavaScript bindings, we depend on the *wasm-bindgen* and *js-sys* crates. *wasm-bindgen* together with *js-sys* allow developers to create JavaScript-compatible WebAssembly code, and with *wasm-pack* we can package this WebAssembly code into JavaScript files that look similar to this one:

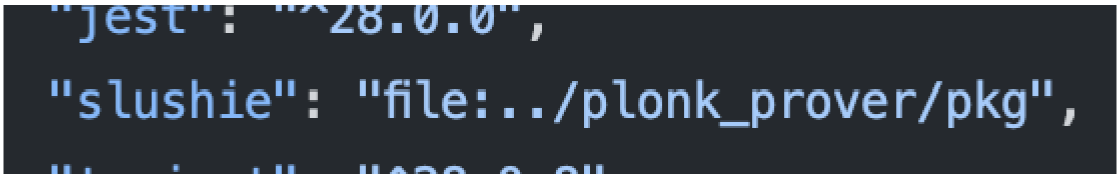


```
const path = require('path').join(__dirname, 'plonk_prover_bg.wasm');
const bytes = require('fs').readFileSync(path);

const wasmModule = new WebAssembly.Module(bytes);
const wasmInstance = new WebAssembly.Instance(wasmModule, imports);
wasm = wasmInstance.exports;
module.exports.__wasm = wasm;
```

Fig. 4.1, the JavaScript package generated by wasm-pack

Lastly, to use it in any JavaScript package, we depend on the generated package in *package.json* of the target package:

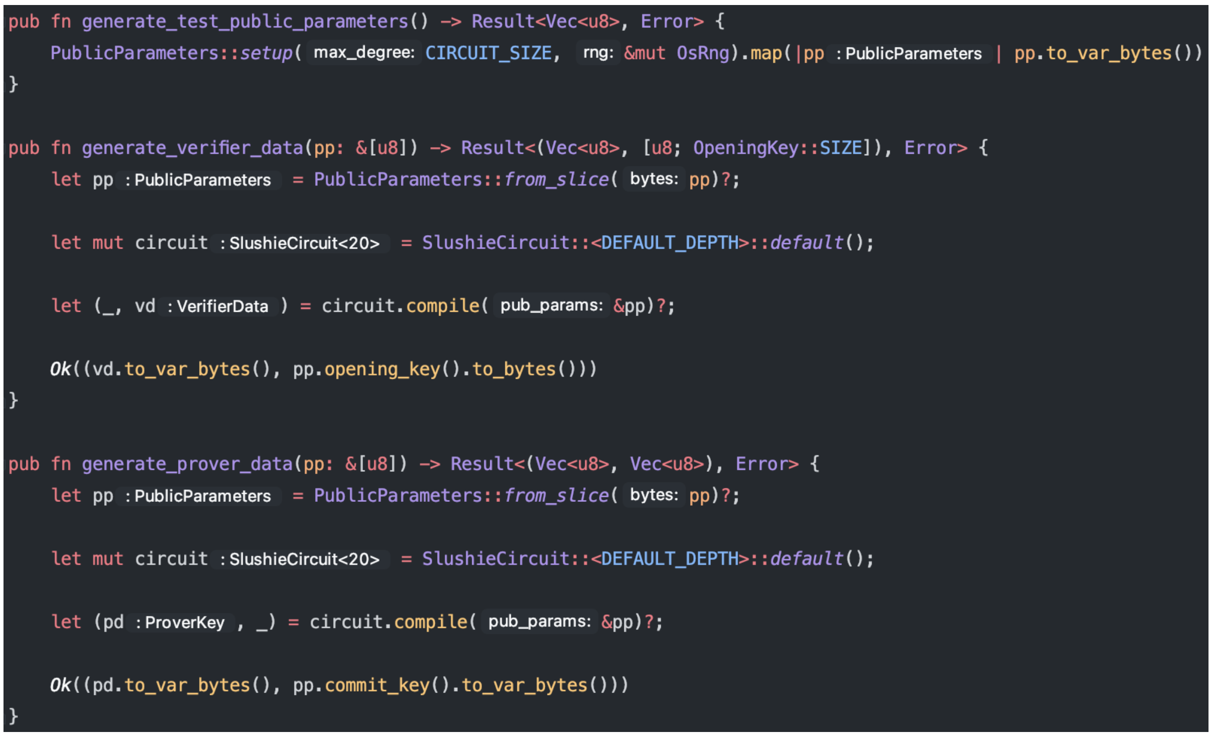


```
  "jest": "^28.0.0",
  "slushie": "file:../plonk_prover/pkg",
```

Fig. 4.2, import of Slushie WebAssembly code, wrapped in a JavaScript package

SRS

As Benarroch states in the ZKProof Community Reference [8, p.58], ZKP schemes require a URS (uniform reference string) or SRS (structured reference string) for their soundness and/or ZK properties. Since Plonk requires an SRS to be used in both proof generation and proof verification processes in some way, we have created utilities that aid in creating the SRS, prover data (the prover key pk, used to construct a zero-knowledge proof, and the commit key ck, used for Plonk's KZG10 commitments), and the verifier data (verifier key, public input indices w.r.t witness positions, and the opening key).



```
pub fn generate_test_public_parameters() -> Result<Vec<u8>, Error> {
    PublicParameters::setup( max_degree: CIRCUIT_SIZE,  rng: &mut OsRng).map(|pp :PublicParameters | pp.to_var_bytes())
}

pub fn generate_verifier_data(pp: &[u8]) -> Result<(Vec<u8>, [u8; OpeningKey::SIZE]), Error> {
    let pp :PublicParameters  = PublicParameters::from_slice( bytes: pp)?;

    let mut circuit :SlushieCircuit<20>  = SlushieCircuit::<DEFAULT_DEPTH>::default();

    let (_, vd :VerifierData ) = circuit.compile( pub_params: &pp)?;

    Ok((vd.to_var_bytes(), pp.opening_key().to_bytes()))
}

pub fn generate_prover_data(pp: &[u8]) -> Result<(Vec<u8>, Vec<u8>), Error> {
    let pp :PublicParameters  = PublicParameters::from_slice( bytes: pp)?;

    let mut circuit :SlushieCircuit<20>  = SlushieCircuit::<DEFAULT_DEPTH>::default();

    let (pd :ProverKey , _) = circuit.compile( pub_params: &pp)?;

    Ok((pd.to_var_bytes(), pp.commit_key().to_var_bytes()))
}
```

Fig. 5, functions that generate SRS for the zk-SNARK

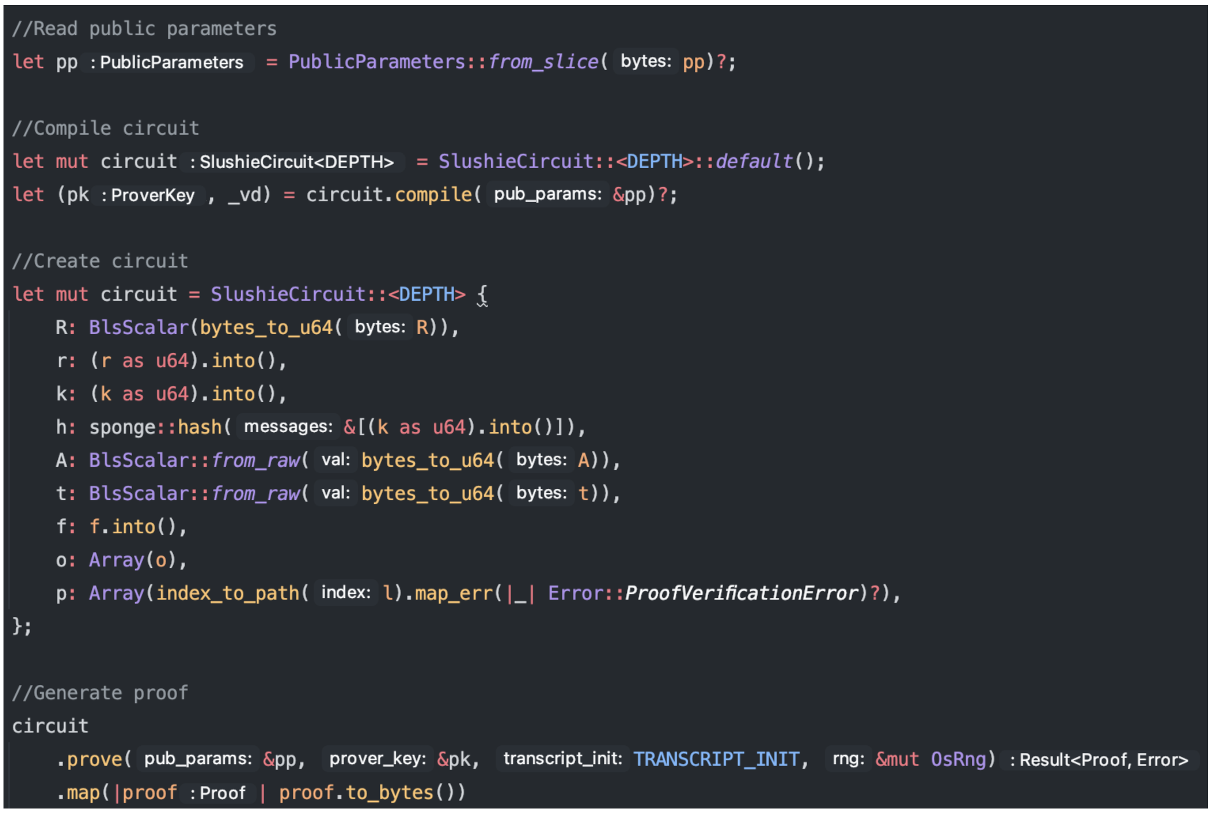
Proof generation & proof verification

In the implementation of $Slushie$, we provide utilities, used for constructing the proofs and verifying the proofs.

In general, the proof generation process consists of four parts. Given a circuit called $SlushieCircuit$, public parameters pp, leaf index l, root R, tree openings o, nullifier k, randomness r, recipient address A, relayer address t, and fee F, we:

1. Deserialize public parameters pp into $PublicParameters$ struct
2. Compile an empty circuit with given public parameters to obtain the prover key
3. Create an instance of the circuit with provided public and private inputs

4. Generate the proof using public parameters pp, prover key pk, transcript initializer $TRANSCRIPT\_INIT$, and a $CSPRNG$ source.



```
//Read public parameters
let pp : PublicParameters  = PublicParameters::from_slice( bytes: pp)?;

//Compile circuit
let mut circuit : SlushieCircuit<DEPTH>  = SlushieCircuit::<DEPTH>::default();
let (pk : ProverKey , _vd) = circuit.compile( pub_params: &pp)?;

//Create circuit
let mut circuit = SlushieCircuit::<DEPTH> {
    R: BlsScalar(bytes_to_u64( bytes: R)),
    r: (r as u64).into(),
    k: (k as u64).into(),
    h: sponge::hash( messages: &[(k as u64).into()]),
    A: BlsScalar::from_raw( val: bytes_to_u64( bytes: A)),
    t: BlsScalar::from_raw( val: bytes_to_u64( bytes: t)),
    f: f.into(),
    o: Array(o),
    p: Array(index_to_path( index: l).map_err(|_| Error::ProofVerificationError)?),
};

//Generate proof
circuit
    .prove( pub_params: &pp,  prover_key: &pk,  transcript_init: TRANSCRIPT_INIT,  rng: &mut OsRng) : Result<Proof, Error>
    .map(|proof : Proof | proof.to_bytes())
```

Fig. 6.1, generation of the zero-knowledge proof with SRS setup

The proof verification process consists of four parts. Given a circuit called $SlushieCircuit$, public parameters pp, root R, nullifier hash h, recipient address A, relayer address t, fee f, and the proof $proof$

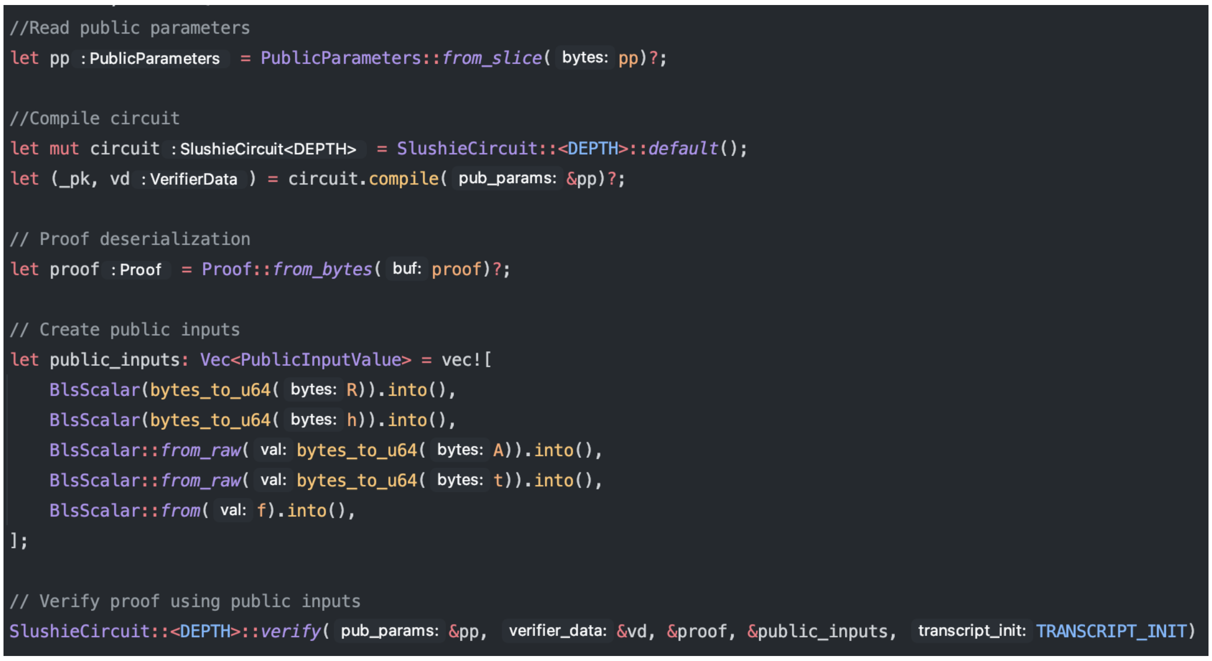
1. Deserialize public parameters pp into $PublicParameters$ struct

2. Compile an empty circuit with given public parameters to obtain the verifier data

3. Deserialize proof $proof$ into the $Proof$ struct

4. Create a vector of public inputs R, h, A, t, f to be passed to the verifying function

5. Verify proof with provided public parameters pp, verifier data obtained in step 2, proof $proof$, the vector of public inputs from step 4, and the $TRANSCRIPT\_INIT$ transcript initializer



```
//Read public parameters
let pp :PublicParameters  = PublicParameters::from_slice( bytes: pp)?;

//Compile circuit
let mut circuit :SlushieCircuit<DEPTH>  = SlushieCircuit::<DEPTH>::default();
let (_pk, vd :VerifierData ) = circuit.compile( pub_params: &pp)?;

// Proof deserialization
let proof :Proof  = Proof::from_bytes( buf: proof)?;

// Create public inputs
let public_inputs: Vec<PublicInputValue> = vec![
    BlsScalar(bytes_to_u64( bytes: R)).into(),
    BlsScalar(bytes_to_u64( bytes: h)).into(),
    BlsScalar::from_raw( val: bytes_to_u64( bytes: A)).into(),
    BlsScalar::from_raw( val: bytes_to_u64( bytes: t)).into(),
    BlsScalar::from( val: f).into(),
];

// Verify proof using public inputs
SlushieCircuit::<DEPTH>::verify( pub_params: &pp,  verifier_data: &vd, &proof, &public_inputs,  transcript_init: TRANSCRIPT_INIT)
```

Fig. 6.2, verification of the zero-knowledge proof with SRS setup

However, due to computational constraints for the prover, we may want to omit the operations of deserializing and constructing the public parameters when generating or verifying proofs. For this reason, we add two functions that don't require passing and deserializing the public parameters, which can exceed $3MB$ for a circuit of size 2^{15} gates. Instead, they use prover data for proof generation and verifier data for proof verification. The motivation for using these variants of proof generation and proof verification is:

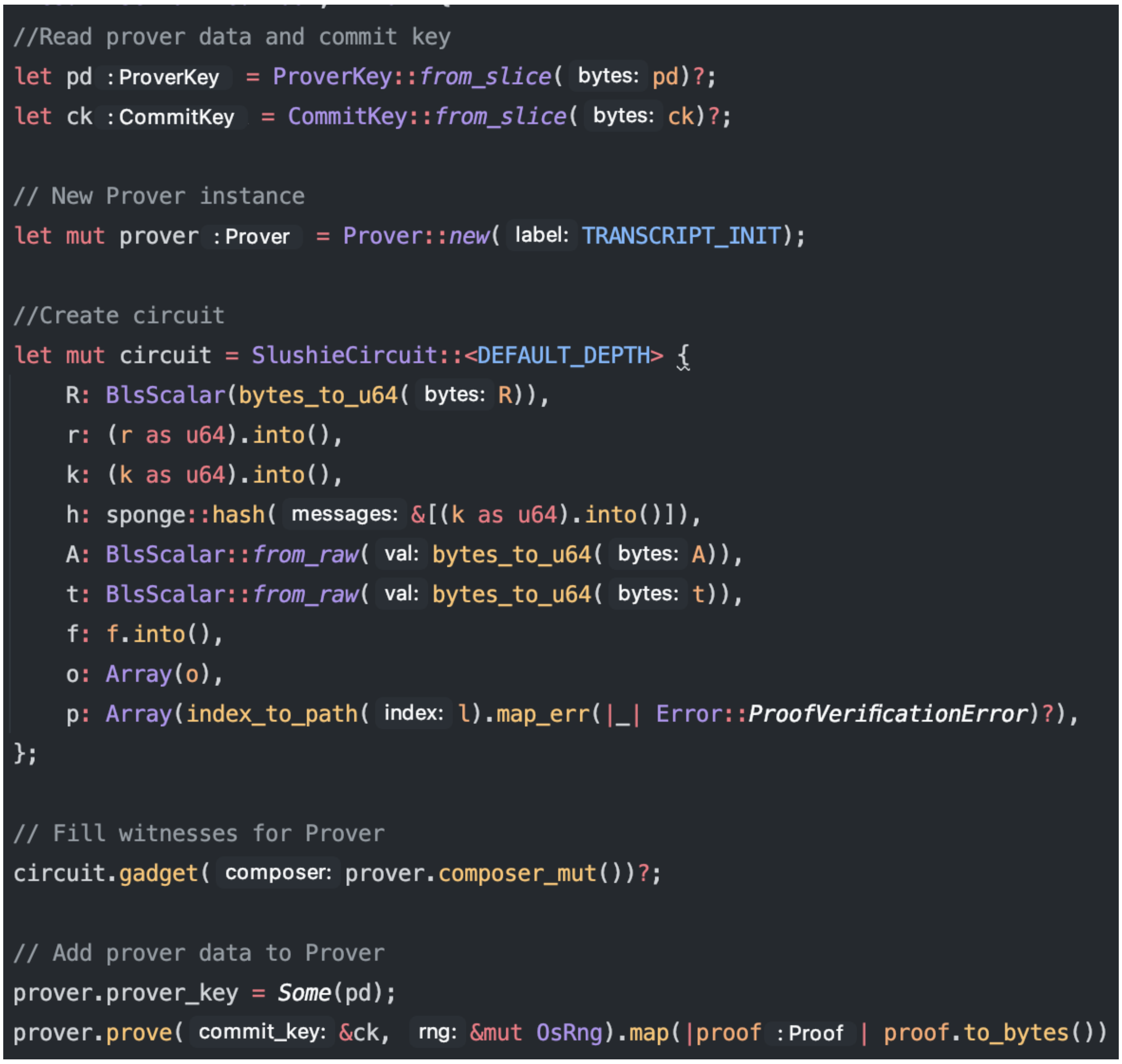
1. For proof generation, generating public parameters can be slow, making the already long proof generation process even longer

2. For proof verification, public parameters of size around $3MB$ can be too large to pass to a smart contract, and deserializing them would consume a lot of gas.

Considering the above reasons, we define the following functions for proof generation and proof verification, where both of them accept prover and verifier data as arguments respectively.

For proof generation, we create the following modification of the proving function, which consists of six parts:

1. Deserialize prover data pd into $ProverKey$ struct,
deserialize commit key ck into $CommitKey$ struct
2. Create a new prover instance with transcript initializer $TRANSCRIPT\_INIT$
3. Create a new circuit instance with appropriate public and private inputs
4. Fill the allocated witnesses for the prover
5. Insert the prover key pd into the prover
6. Generate the proof using the deserialized commit key ck and a $CSPRNG$ source.



```
//Read prover data and commit key
let pd :ProverKey = ProverKey::from_slice( bytes: pd)?;
let ck :CommitKey = CommitKey::from_slice( bytes: ck)?;

// New Prover instance
let mut prover :Prover = Prover::new( label: TRANSCRIPT_INIT);

//Create circuit
let mut circuit = SlushieCircuit::<DEFAULT_DEPTH> {
    R: BlsScalar(bytes_to_u64( bytes: R)),
    r: (r as u64).into(),
    k: (k as u64).into(),
    h: sponge::hash( messages: &[(k as u64).into()]),
    A: BlsScalar::from_raw( val: bytes_to_u64( bytes: A)),
    t: BlsScalar::from_raw( val: bytes_to_u64( bytes: t)),
    f: f.into(),
    o: Array(o),
    p: Array(index_to_path( index: l).map_err(|_| Error::ProofVerificationError)?),
};

// Fill witnesses for Prover
circuit.gadget( composer: prover.composer_mut())?;

// Add prover data to Prover
prover.prover_key = Some(pd);
prover.prove( commit_key: &ck,  rng: &mut OsRng).map(|proof :Proof | proof.to_bytes())
```

Fig. 6.3, generation of the zero-knowledge proof without SRS setup

For proof verification, we create the following modification of the verifying function, which consists of five parts:

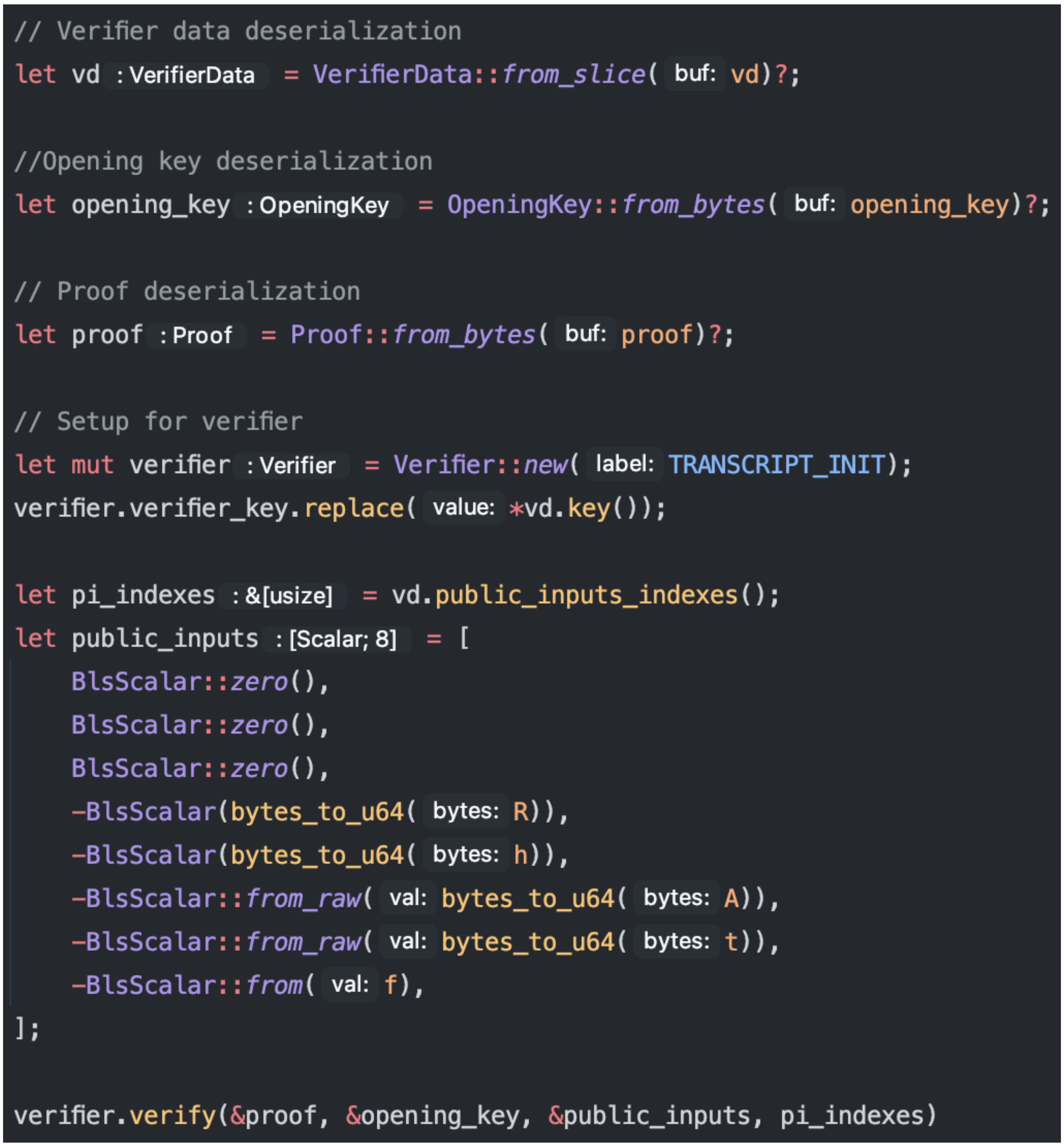
1. Deserialize the verifier data vd into $VerifierData$ struct, deserialize the opening key $opening\_key$ into $OpeningKey$ struct

2. Deserialize the supplied proof $proof$ into the $Proof$ struct

3. Create a new verifier instance and replace the verifier key with deserialized verifier data vd from step 1

4. Construct the public input vector with the supplied parameters

5. Verify the proof $proof$ using the opening key $opening\_key$ against the public input vector from step 4.



```
// Verifier data deserialization
let vd : VerifierData = VerifierData::from_slice( buf: vd)?;

//Opening key deserialization
let opening_key : OpeningKey = OpeningKey::from_bytes( buf: opening_key)?;

// Proof deserialization
let proof : Proof = Proof::from_bytes( buf: proof)?;

// Setup for verifier
let mut verifier : Verifier = Verifier::new( label: TRANSCRIPT_INIT);
verifier.verifier_key.replace( value: *vd.key());

let pi_indexes : &[usize] = vd.public_inputs_indexes();
let public_inputs : [Scalar; 8] = [
    BlsScalar::zero(),
    BlsScalar::zero(),
    BlsScalar::zero(),
    -BlsScalar(bytes_to_u64( bytes: R)),
    -BlsScalar(bytes_to_u64( bytes: h)),
    -BlsScalar::from_raw( val: bytes_to_u64( bytes: A)),
    -BlsScalar::from_raw( val: bytes_to_u64( bytes: t)),
    -BlsScalar::from( val: f),
];

verifier.verify(&proof, &opening_key, &public_inputs, pi_indexes)
```

Fig. 6.4, verification of the zero-knowledge proof without SRS setup

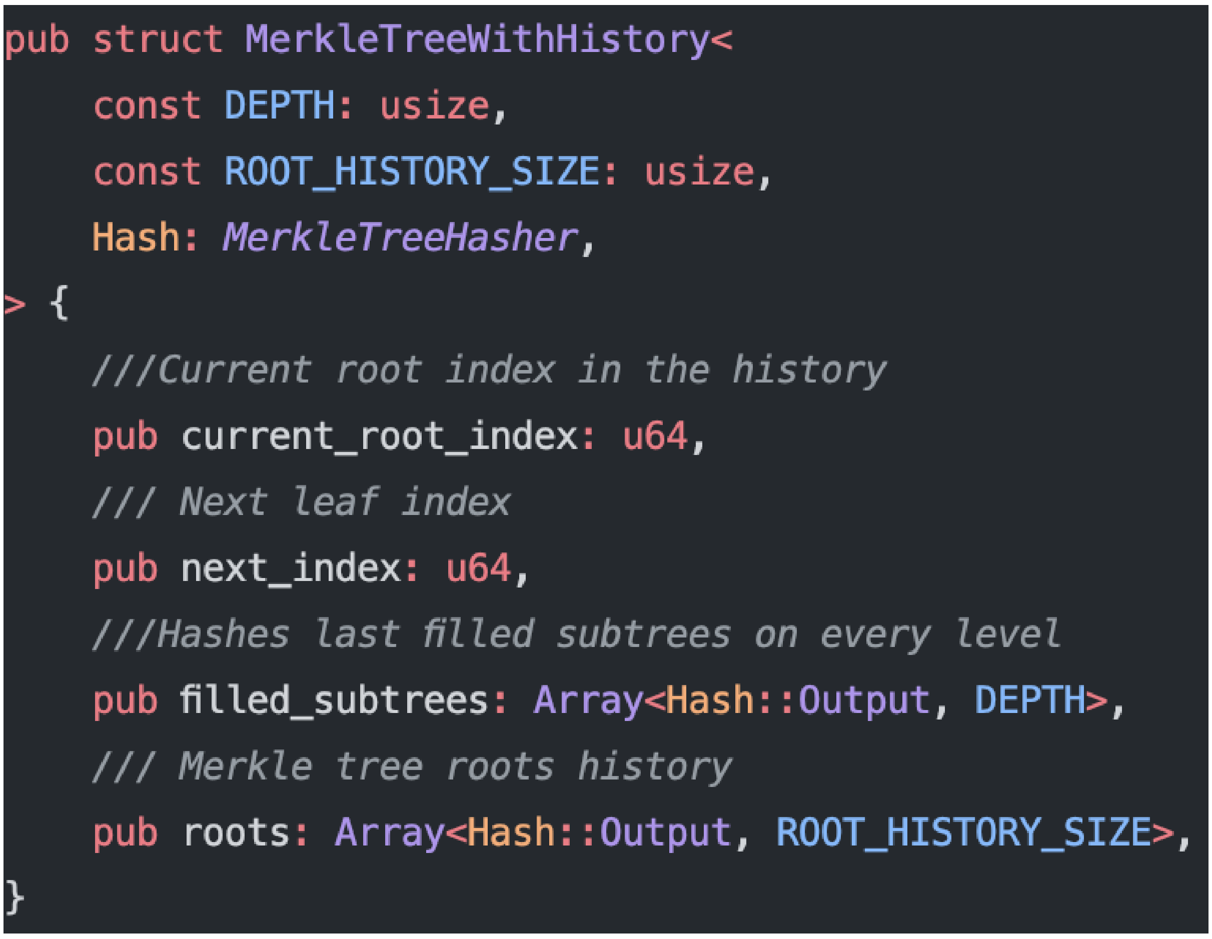
Merkle trees

As we noted before, in *plonk_prover* we define two implementations of the Merkle trees.

Merkle tree with history

In general, Merkle trees are used to store some important "state" of the system. With $Slushie$, Merkle trees are used to store the information about users' commitments to deposits, to construct proofs for any given commitment and root, and to verify the supplied proofs when the $withdraw$ transaction is called.

Pertsev et al. proposed a modification to the original Merkle tree, which they called a Merkle tree with history. In essence, instead of storing all nodes of the tree, the Merkle tree with history stores the current root index, the next root index, the last filled subtrees on every level, and an array of roots of size $ROOT\_HISTORY\_SIZE$. These modifications make the Merkle tree efficient enough storage-wise to store in a smart contract, yet in conjunction with blockchain events that emit stored commitments, which are inserted into the Merkle tree, one can recreate the Merkle tree fully at each point in time. Our implementation of the Merkle tree is a direct port from Solidity, though it contains some different Rust-specific abstractions. These include specifying the $DEPTH$ and $ROOT\_HISTORY\_SIZE$ parameters as constant generics, while also parametrizing over the *Hash* type (the hashers).

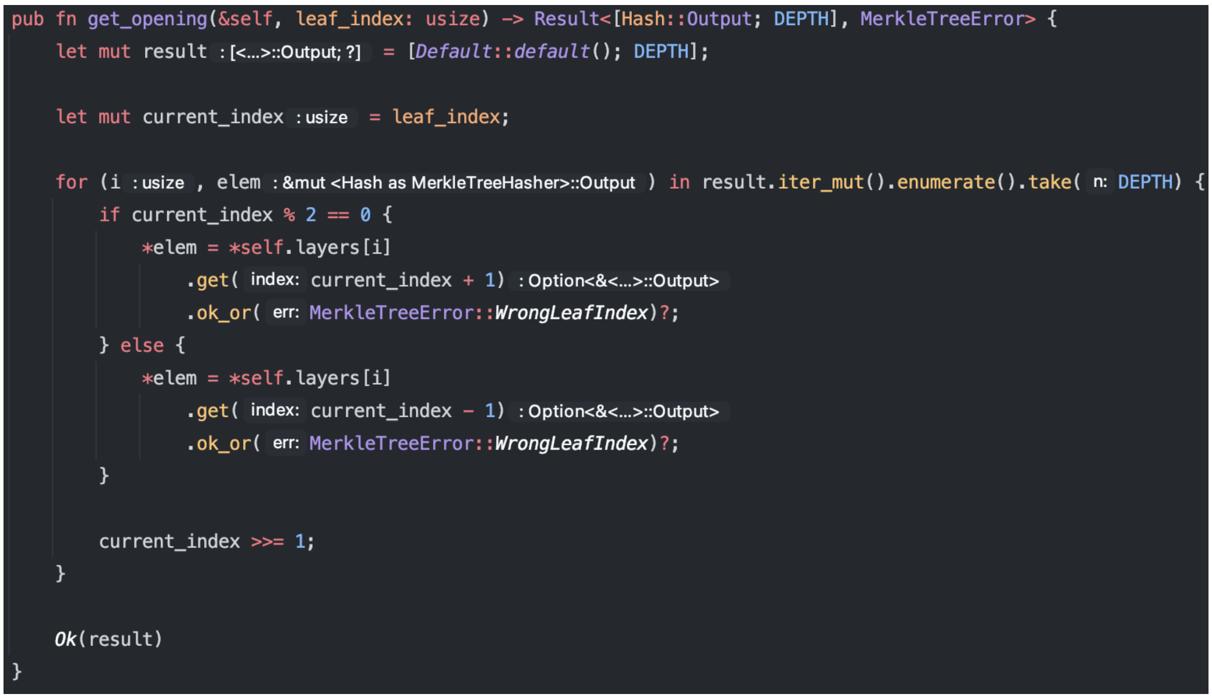


```rust
pub struct MerkleTreeWithHistory<
    const DEPTH: usize,
    const ROOT_HISTORY_SIZE: usize,
    Hash: MerkleTreeHasher,
> {
    ///Current root index in the history
    pub current_root_index: u64,
    /// Next leaf index
    pub next_index: u64,
    ///Hashes last filled subtrees on every level
    pub filled_subtrees: Array<Hash::Output, DEPTH>,
    /// Merkle tree roots history
    pub roots: Array<Hash::Output, ROOT_HISTORY_SIZE>,
}
```

Fig. 7.1, Merkle tree with history structure

Flat Merkle tree

The "flat" Merkle tree is a different implementation of the Merkle tree. As opposed to the Merkle tree with history, the flat Merkle tree stores all nodes of the Merkle tree. It is primarily used to get the openings of the Merkle tree:



```
pub fn get_opening(&self, leaf_index: usize) -> Result<[Hash::Output; DEPTH], MerkleTreeError> {
    let mut result :[<...>::Output; ?]  = [Default::default(); DEPTH];

    let mut current_index :usize  = leaf_index;

    for (i :usize , elem :&mut <Hash as MerkleTreeHasher>::Output ) in result.iter_mut().enumerate().take( n: DEPTH) {
        if current_index % 2 == 0 {
            *elem = *self.layers[i]
                .get( index: current_index + 1) :Option<&<...>::Output>
                .ok_or( err: MerkleTreeError::WrongLeafIndex)?;
        } else {
            *elem = *self.layers[i]
                .get( index: current_index - 1) :Option<&<...>::Output>
                .ok_or( err: MerkleTreeError::WrongLeafIndex)?;
        }

        current_index >>= 1;
    }

    Ok(result)
}
```

Fig. 7.2, opening generation function

Plonk Prover CLI

The Plonk Prover CLI is a crate for off-chain operations with commitments and proofs. It uses the *Plonk prover* internally, compiles to native target code, and ensures the usage of exactly the same commitment generation, proof generation, and verification logic on the users' local machines and in the smart contract. It is worth noting that the CLI is an essential part of the whole workflow of depositing funds into *Slushie* and withdrawing them. However, as we possess the ability to cross-compile our core logic in *plonk prover* to JavaScript + WebAssembly, the inclusion of this CLI logic into a front-end library would be beneficial for the overall user experience. We consider it as a future improvement to our work.

The CLI structure

We use *clap* as the CLI library. The library allows us to define statically typed command types in the *commands.rs* file, to which we add the implementations in the *actions.rs* file. Other than that, since this crate is a binary crate, we define the CLI startup logic in the *main.rs* file, and define any useful utilities in the *utils.rs* file.

The commands

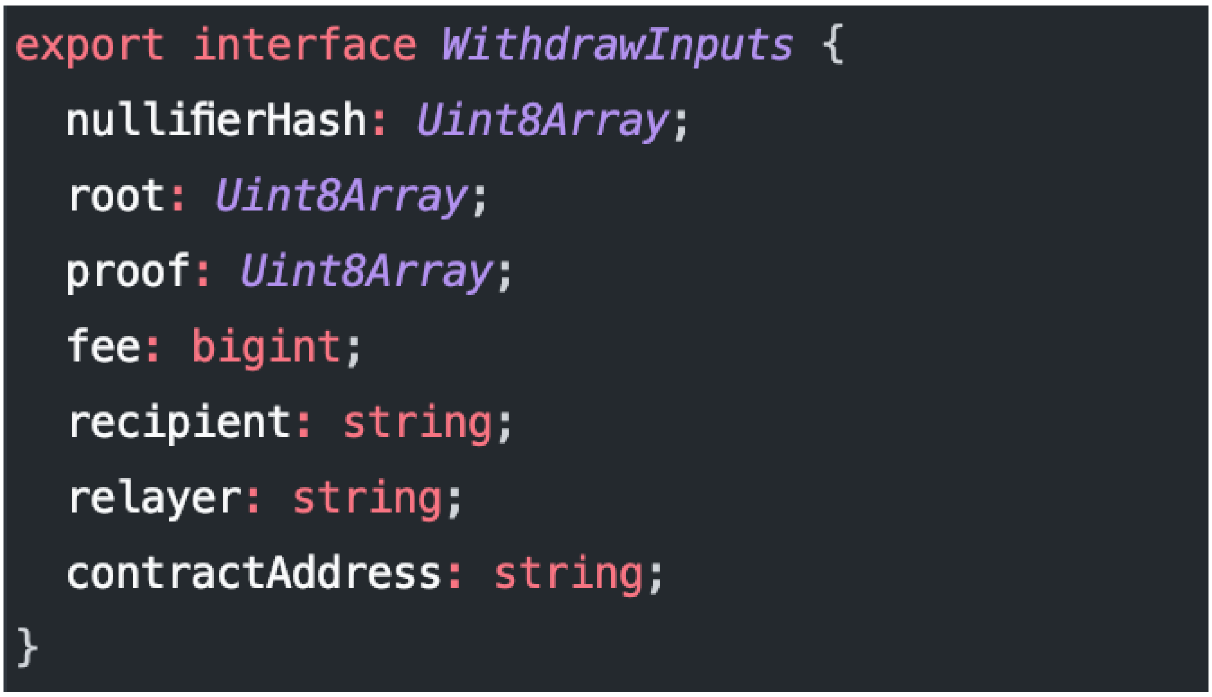
The CLI supports the following commands:

1. *GenerateCommitment* – allows the user of the CLI to generate a new random commitment that consists of the nullifier, the randomness, and the commitment itself (commitment generation described in the *Slushie (smart contract)* section).
2. *GenerateTestPublicParameters* – generates an SRS for producing proofs, prover data, and verifier data.
3. *GenerateVerifierData* – generates the verifier data from the SRS, producing the serialized verifier data and the opening key for KZG10 commitments.
4. *GenerateProverData* – generates the prover data from the SRS, producing the serialized prover data and the commit key for KZG10 commitments.
5. *GenerateProof* – generates the proof from a set of inputs described in the *Proof generation & proof verification* section.

For bigger files, like tree openings or the SRS, we allow providing a local path to those files.

Relayer

The relayer is a Typescript application that contains an implementation of an HTTP server with one endpoint, "/withdraw". This endpoint accepts JSON-formatted data, relevant to the *withdraw* transaction. This includes the nullifier hash, the Merkle tree root, the serialized proof, the relayer fee, the recipient address, the relayer address, and the Slushie contract address.



```
export interface WithdrawInputs {
  nullifierHash: Uint8Array;
  root: Uint8Array;
  proof: Uint8Array;
  fee: bigint;
  recipient: string;
  relayer: string;
  contractAddress: string;
}
```

Fig. 8, WithdrawInputs request payload structure

As a future improvement, the request payload must be constructed on the front-end. However, in the scope of this work, this is done manually, and the request is sent using *cURL*.

A note on the implementation of the relayer

Initially, we planned to implement the relayer as a Rust-based JSON-RPC server. However, after performing a critical dependency update, one vital library that was used to submit transactions to the *Substrate*-based chains broke the process of generating signed extrinsics, which we use to perform withdrawals. After a thorough, yet unsuccessful investigation, it became evident that this issue cannot be fixed without creating a local clone of that library.

Instead, we implemented the relayer using Typescript and a Typescript-based library for working with *Substrate*-based chains. The development proved to be much quicker, measuring in hours and not days. To add to that, this implementation turned out to be more reliable due to the relevant libraries being actively maintained.

Conclusion

In this work, we explored the possibility of implementing a zero-knowledge proof-based cryptocurrency mixer in the Substrate ecosystem. As a result, we successfully built an *ink!* smart contract and some necessary tooling around it. Other than implementing a copy of on-chain logic from [5], we have also improved the logical integrity of the tooling, reusing exactly the same code for working with zero-knowledge proofs in the smart contract and in the offchain prover CLI.

Although implementing this mixer turned out to be successful, *Substrate* still lacks some features that would make Slushie ready for production use. The main problem which limited our ability to only work with one commitment at a time was the lack of an equivalent of *eth_getLogs* RPC in *Substrate* nodes. We need it to construct the openings of the Merkle tree for proof generation, which is not possible with the current *Substrate* implementation.

We achieved a proof of concept level of readiness with this project. In the future, we will implement the improvements listed in this work, and Slushie will contribute to the privacy of users in *pallet-contracts*-compatible networks.

References

[1] Pertsev, A., Semenov, R., & Storm, R. (2019, December 17). *Tornado Cash Privacy Solution Version 1.4*. Tornado Cash Privacy Solution Version 1.4. Retrieved April 19, 2023, from https://berkeley-defi.github.io/assets/material/Tornado%20Cash%20Whitepaper.pdf

[2] Gabizon, A., Williamson, Z. J., & Ciobotaru, O. (2022, August 17). *PlonK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge[0.72cm]*. Cryptology ePrint Archive. Retrieved April 19, 2023, from https://eprint.iacr.org/2019/953.pdf

[3] Grassi, L., Khovratovich, D., Rechberger, C., Roy, A., & Schofnegger, M. (n.d.). *POSEIDON: A New Hash Function for Zero-Knowledge Proof Systems (Updated Version)*. Cryptology ePrint Archive. Retrieved April 19, 2023, from https://eprint.iacr.org/2019/458.pdf

[4] Wood, G. (2022, October 24). *ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER*. GitHub Pages. Retrieved April 24, 2023, from https://ethereum.github.io/yellowpaper/paper.pdf

[5] Tornado.Cash. (n.d.). *tornado-core*. https://github.com/tornado-repositories/tornado-core. Retrieved April 19, 2023, from https://github.com/tornado-repositories/tornado-core/tree/master

[6] Dusk Network B.V. (n.d.). *Pure Rust implementation of the PLONK ZKProof System done by the Dusk-Network team*. GitHub. Retrieved April 19, 2023, from https://github.com/dusk-network/plonk

[7] Gennaro, R., Gentry, C., Parno, B., & Raykova, M. (2012). *Quadratic Span Programs and Succinct NIZKs without PCPs*. Cryptology ePrint Archive. Retrieved April 19, 2023, from https://eprint.iacr.org/2012/215.pdf

[8] Benarroch, D. (2022, July 17). *ZKProof Community Reference*. ZKProof Resources. Retrieved April 19, 2023, from https://docs.zkproof.org/pages/reference/reference.pdf