Nightfall

Chaitanya Konda, Michael Connor, Duncan Westland,
Quentin Drouot, Paul Brody

EY Global Blockchain R&D

2019 May

Contents

I Introduction 2

1 zk-SNARKs 2

II The Application 5

2 ZoKrates 6

3 Trusted Setup 7

4 Smart Contracts 10

5 Microservices 15

6 Finite Fields and Bit Lengths 19

III The Protocols 22

7 ERC-721 (non-fungible) tokens 23

8 ERC-20 (fungible) Tokens 44

Document prepared by Michael Connor

Part I

Introduction

Table of Contents

1 zk-SNARKs **2**
1.1 Overview . . . 2
1.2 Background . . . 3
1.3 Efficiency . . . 3
1.4 Proof generation . . . 4

In this document we discuss Nightfall [1] - an open source suite of tools designed to enable private token transactions over the public Ethereum blockchain.

Nightfall is currently compatible with smart contracts which adhere to either of Ethereum's ERC-20 or ERC-721 token standards. These token standards have been chosen because of their prevalence in the Ethereum community today. Nightfall could indeed be expanded in the future to support other token standards.

ERC-20 tokens are fungible in the sense that they can be subdivided and individual units are interchangeable.
ERC-721 tokens are non-fungible in the sense that they represent something unique.

Nightfall was created by EY and released into the public domain in May 2019.

1 zk-SNARKs

1.1 Overview

A zero-knowledge succinct non-interactive argument of knowledge (zk-SNARK) enables a 'Prover' to demonstrate that they have correctly performed a calculation on a set of inputs, without revealing some of those inputs. This is useful in the context of privately transacting on a public blockchain. In a 'traditional' (non-private, ERC-20) blockchain transfer, computations to update a sender and receiver's balances are performed publicly 'on-chain' within the `transfer` function of a smart contract. This public `transfer` function requires the `to`, and `amount` values to be publicly input into the calculation.

Figure 1 depicts a public token transfer over Ethereum.

```
/**
* @dev Transfer token for a specified address
* @param _to The address to transfer to.
* @param _value The amount to be transferred.
*/
function _transfer(address _to, uint256 _value) internal returns (bool) {
  require(_value <= balances[msg.sender]);
  require(_to != address(0));

  balances[msg.sender] = balances[msg.sender].sub(_value);
  balances[_to] = balances[_to].add(_value);
  emit Transfer(msg.sender, _to, _value);
  return true;
}
```

Figure 1: An implementation of the ERC-20 'transfer' function

For private transfers, we require that the `to` and `amount` values to be kept private from the blockchain. This is non-trivial, because for any 'traditional' computation within a smart contract, any values which are used within a calculation must necessarily be made public in order for all nodes to agree on the new states of the smart contract. It therefore follows that we need to design alternative computations which are performed on-chain, in order to hide the inputs to a `transfer`. We will also need to re-think how 'ownership' is ascribed to tokens. In a traditional ERC-20 contract the `balances` of each

Ethereum address are public mappings, which is incompatible with our notion of privacy. We require that the public key receiving tokens is not revealed by the transfer.

In order to keep the `to` and `amount` inputs private between the sender and receiver we use zk-SNARKs. The sender (or 'Prover') runs a computation *privately* on their own computer. They pass *private* inputs into this computation and get a set of *public* outputs which they share with the blockchain. The public outputs appear as unreadable encrypted values to all observers; only the sender and receiver can interpret their full meaning. In order for these encrypted values to have 'meaning' to all observers, the Prover also shares with the blockchain a corresponding 'proof' of having correctly computed these outputs. Together this proof and these public outputs can be verified in such a way that everyone can be convinced that a pre-agreed calculation has been performed on a particular set of private inputs to produce the public outputs. In this case, the pre-agreed calculation represents a 'transfer', and verification of the proof and public outputs can be unambiguously interpreted by observers as *"somebody has submitted a binding intention to transfer funds to someone else"*. For full details on the Prover's computation and the public outputs submitted to the blockchain, see Part III.

1.2 Background

Nightfall leverages the impressive mathematics of zk-SNARKs to achieve privacy. In this paper zk-SNARKs are used in a blackbox manner and we do not discuss how they work. We encourage interested readers to refer to explanations by Zcash [2], Buterin [3], and Reitwiessner[4].

Nightfall is implemented under a zk-SNARK by Groth and Maller [5]. This zk-SNARK is simulation extractable, meaning that it is hard for an attacker to gain an advantage from other zk-SNARKs on the blockchain. It has highly competitive efficiency in terms of proof size and verifier time. On the other hand, it's security relies on the 'Knowledge-of-exponent' assumption, which is a non-standard cryptographic assumption (in other words, although it is currently considered cryptographically secure, it is more likely to be broken than RSA or discrete-log).

1.3 Efficiency

Nightfall has a proving system for mint, transfer, and burn algorithms for both fungible and non-fungible assets which are proven secure using a zk-SNARK. These protocols are described in Part III. In Table 1 we give an overview of the costs of each of these algorithms.

Algorithm	Constraints	Key Sizes		zk-SNARK Size		Costs[4]	
		Proving Key[1] (Mbytes)	Verification Key[2] (bytes)	Input[3] (bytes)	Proof[3] (bytes)	Gas (M)	US $
`ft-mint`	83,000	32	414	128	256	1.8	10.21
`ft-transfer`	2,292,000	859	637	352	256	2.7	14.77
`ft-burn`	1,075,000	401	478	192	256	1.7	9.49
`nft-mint`	83,000	32	510	160	256	1.9	10.60
`nft-transfer`	1,158,000	433	446	224	256	2.1	11.52
`nft-burn`	1,075,000	401	510	224	256	1.8	10.03

Table 1: Efficiency overview of Nightfall.

[1] As calculated within ZoKrates during the 'setup' stage. Files stored on a User's hard drive will be larger.

[2] As calculated within ZoKrates during the 'setup' stage. Each elliptic curve point of a Verification Key is encoded as either two `uint256` values (G1) or four `uint256` values (G2) within the Nightfall smart contracts, and so might be larger than shown here.

[3] Based on their sizes when passed as calldata to the Nightfall smart contracts. Each input is a `uint256`. Each elliptic curve point of a Proof is encoded as either two `uint256` values (G1) or four `uint256` values (G2).

[4] The price of Ether as at 30 May 2019 is $278 (according to Coin Market Cap). Costs are based on a gas price of 20Gwei.
Gas measurements have been taken over an entire transaction. I.e. the gas cost shown for an nft-transfer (for example) reflects the overall cost of computation and storage across *all* contracts involved in that transaction.
Verification of the zk-SNARK (within the Verifier contract) forms only a part of this total gas cost. The gas costs for the verification of a zk-SNARK *alone* (within the Verifier contract) have not been measured, due to the difficulty in attributing the final 'storage refund' at the end of a transaction between the Shield contract and the Verifier contract.

1.4 Proof generation

Nightfall uses ZoKrates to generate proofs. With that, the time it takes to generate a 'proof' depends mainly on:

1. the complexity of the computation being 'proven'
2. the computing resources being allocated to generate the proof
3. advances in ZoKrates' efficiency for generating proofs

1. With zk-SNARKs, a 'computation' is deconstructed into a series of arithmetic constraints. In Nightfall, the most constraints come from doing sha256 hashing from the leaf of a Merkle Tree to its root (see Part III, the Protocols, for details). This repeated sha256 hashing is performed during a 'transfer' and a 'burn' of both fungible and non-fungible token commitments. Most notably, for a fungible transfer, we need to perform this repeated hashing from leaf-to-root twice; for two leaves.

At the moment, we have configured Nightfall to have a Merkle Tree of 'depth' 33. That is, we have chosen to have a huge Merkle Tree with $2^{32} = 4,294,967,296$ leaves. That's capacity for over 4 billion token commitments before the Shield Contract runs out of space. If your use-case doesn't need to accommodate so many commitments for the rest of time, then you could reduce the depth of the tree in your implementation.

A different hashing function would also reduce the time to generate a proof. Pedersen commitments were considered, because they have far fewer constraints. However the repeated hashing from the leaf of a Merkle Tree to its root must also be performed *on chain* (within the Shield Contract) with Nightfall. Given that there are no precompile contracts which reduce the gas costs for Pedersen commitment calculation, it is *significantly* cheaper to use sha256.

2. The proof for a fungible transfer takes the most time to generate. Generally this can take anywhere from 'a few minutes' to 'around 10 minutes', depending on the power of the computer.

3. ZoKrates is continually updated with efficiency improvements. A newer image of ZoKrates could perhaps improve upon Nightfall's current proving times.

Part II

The Application

Table of Contents

2 ZoKrates **6**

2.1 ZoKrates JavaScript Wrapper . . . 6

3 Trusted Setup **7**

4 Smart Contracts **10**

4.1 Pre-Existing Contracts . . . 10

4.2 Shield contracts . . . 12

4.3 Verifier contracts . . . 13

4.4 Verifier Registry contracts . . . 14

4.5 Deployment of Contracts . . . 14

5 Microservices **15**

5.1 zkp . . . 15

5.2 offchain . . . 16

5.3 accounts . . . 17

5.4 database . . . 17

5.5 ui . . . 17

6 Finite Fields and Bit Lengths **19**

2 ZoKrates

Nightfall uses ZoKrates [6] to generate zk-SNARKs. As discussed in Part I, zk-SNARKs allow a Prover to 'prove' that they have performed a particular computation, and in order to do so they must convert that computation into an abstract 'proof'. ZoKrates gives us a domain specific language (DSL) through which we can express our computations in a human-readable way. It then abstracts the code into the format expected by the zk-SNARK prover and verifier. In all, ZoKrates assists Nightfall with the following:

- A human-readable language for writing code (computations) which can be turned into a zk-SNARK;
- Compilation of human-readable code into constraints;
- Generation of a (verification key, proving key) pair, which together represent the constraints;
- Computation of a 'witness'. A Prover can feed their private inputs and public inputs into ZoKrates, and ZoKrates will produce a 'witness' – a step-by-step vector of evidence that each constraint has been satisfied by these inputs.
- Generation of a 'proof' – This, combined with the public inputs, is the attestation that the constraints of the computation have been satisfied.

2.1 ZoKrates JavaScript Wrapper

Nightfall includes a JavaScript wrapper for each of the ZoKrates functions:

- compile;
- setup;
- compute-witness;
- generate-proof;
- export-verifier.

Nightfall uses the GM17 backend [5] of ZoKrates. Nightfall does not currently support the PGHR13 backend [7]. Nightfall could be adapted to also support PGHR13.
The ZoKrates JavaScript wrapper:

- Uses a Docker Image of ZoKrates from January 2019;
- Performs a Trusted Setup:
  - Mounts '.code' files into a ZoKrates container;
  - Compiles this code into constraints;
  - Performs the 'trusted setup' to output a 'proving key' and a 'verification key'
  - Outputs this 'proving key' and 'verification key' into the mounter directory of the user's local machine;
  - Jsonifies the 'verification key'.
- Generates proofs:
  - Generates a proof for a particular set of public and private inputs (passed from the UI):
  - Extracts the proof object from the container's console (ready for use by the node.js application)

Nightfall doesn't use the hard-coded `verifier.sol` contracts which are created by ZoKrates. Instead, Nightfall uses a single verifier contract which can handle all GM17 verification keys, and all proof submissions against these verification keys. This verifier contract (called `GM17.sol`) adheres to the draft EIP1922 standard.

Where to look?

https://github.com/Zokrates/ZoKrates	ZoKrates source code
https://zokrates.github.io	ZoKrates documentation
`./zkp/src/zokrates.js`	ZoKrates JavaScript Wrapper
`./zkp/code/gm17/`	`.pcode` files
`./zkp/code/README-tools-code-preprop.md`	explanation of `.pcode` syntax
`./zkp/code/README-tools-trusted-setup.md`	how to automatically do the trusted setup
https://github.com/EYBlockchain/zokrates-preprocessor	how to manually transpile from `.pcode` to `.code`
`./zkp/code/README-manual-trusted-setup.md`	how to manually do the trusted setup
`./zkp/contracts/GM17.sol`	for the EIP1922 Verifier contract.
http://eips.ethereum.org/EIPS/eip-1922	for the draft zk-SNARK Verifier Standard.

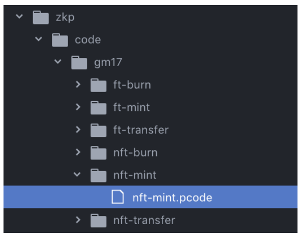
SECURITY WARNINGS

- The Docker Image of ZoKrates from January 2019 is already outdated. It might include security bugs which have since been fixed over in the ZoKrates repository. The syntax of the ZoKrates DSL has also changed considerably since January 2019, and therefore the `.pcode` files of Nightfall are written in outdated syntax.

Figure 2: Security warning: ZoKrates versioning

3 Trusted Setup

A trusted setup is required before anyone can generate a zk-SNARK ('proof') for a particular computation. In Nightfall, there are currently six separate computations for which a trusted setup is required: non-fungible minting, transferring and burning; and fungible minting, transferring and burning. For each computation, the trusted setup is only performed once; at "the beginning of time"; by a generous trusted benefactor; and before the Shield contract can be deployed to the Ethereum blockchain for people to use. In Nightfall it is currently assumed that the generous benefactor will not use secret information obtained in the setup in order to subvert the security of the protocol.



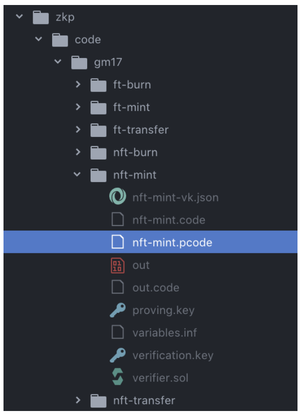


Figure 3: Files in $\mathcal{T}$'s local repository before (left) and after (right) performing a trusted setup.

The trusted setup abstracts the human-readable code of the ZoKrates DSL (files with a `.code` or `.pcode` extension) into a (proving key, verification key) pair.

Suppose $\mathcal{T}$ is a 'trusted benefactor' who intends to use Nightfall to set up the infrastructure which will allow anyone to transfer ownership of tokens under zero knowledge. When $\mathcal{T}$ first clones the Nightfall repository, he only has human-readable computations written in '`.pcode`' syntax. Taking the `./zkp/code/gm17/nft-mint/` folder as an example, $\mathcal{T}$ will initially only have '`nft-mint.pcode`'. The trusted setup will provide $\mathcal{T}$ with: '`nft-mint-vk.json`', '`nft-mint.code`', '`out`', '`out.code`', '`proving.key`', '`variables.inf`', '`verification.key`', '`verifier.sol`'. This is shown in Figure 3 and a brief explanation of each of these files is provided in Table 2.

Trusted Setup Output	Explanation
`nft-mint-vk.json`	The verification key for an 'nft-mint'. This will be stored on-chain, within the Verifier Registry. Every time a user submits a (proof, public inputs) pair to the Shield contract, this pair is verified with respect to the verification key within the Verifier contract.
`nft-mint.code`	Human-readable computation for an 'nft-mint', written in the DSL of ZoKrates.
`nft-mint.pcode`	An abbreviation of the `.code` syntax, for easier writing.
`out`	Ignore.
`out.code`	Ignore
`proving.key`	Used to generate proofs. Every time a User generates a new proof, this file is used by ZoKrates.
`variables.inf`	Ignore
`verification.key`	A representation of the verification key. Nightfall uses the jsonified version of the verification key (`mint-nft-vk.json`) and submits it as a flattened array to the Verifier Registry.
`verifier.sol`	Ignore. This is an example implementation of a verifier contract with the verification key hard-coded into it. It is unused by Nightfall.

Table 2: The files output by the trusted setup.

Remark: The proving keys are by far the largest files required by Users of Nightfall:

nft-mint	77 MB
nft-transfer	1.1 GB
nft-burn	1.0 GB
ft-mint	77 MB
ft-transfer	2.1 GB
ft-burn	1.0 GB

The file sizes shown here reflect the sizes of the files once extracted from a ZoKrates container and stored on a User's local machine. (As opposed to the proving key sizes displayed in Table 1, which are calculated at setup time). The 'transfer' and 'burn' proving keys are particularly large, because of how a User proves that their token commitment exists as a leaf of the on-chain Merkle Tree (see The Protocols). As a default size, the on-chain Merkle Tree is 33-deep, meaning 32 sha256 hashes are performed to calculate the root of the Merkle Tree from the relevant leaf. Each sha256 hash requires around $25,000$ constraints. For a fungible transfer, 64 sha256 hashes are performed (2×32).

Once $\mathcal{T}$ has completed the trusted setup for each of the six computations, he is ready to create the rest of the Nightfall infrastructure. We outline his steps in Figure 4. The Smart Contracts being alluded to are discussed in more detail in the Smart Contracts section.

Notes for a User

Ordinary Users of a Nightfall infrastructure do not need to (and should not!) perform a trusted setup themselves. Only the original creator of the Shield contracts needs to. The trusted setup involves a source of randomness and the (proving key, verification key) pair for a given computation will change each time the trusted setup is performed. Therefore, if a User wishes to generate 'proofs' (zk-SNARKs) to be verified against a verification key which has been stored on the Ethereum mainnet, they should use the exact proving key and vkId which was generated by the trusted setup and shared with everyone.

To explain further, note that for each verification key stored on-chain, there is a corresponding and unique proving key which was generated at the same time, from the same randomness. It is this proving key which Users must use to generate proofs. Using any other key the User's proofs will not verify against the verification key which has been stored on-chain. If a User wishes to generate a proof against an existing, already-deployed verification key, they will need to request the corresponding proving key from the creator of the verification key.

SECURITY WARNINGS

- Performing the initial 'trusted setup' of a computation – to convert a `.code` file into a (proving key, verification key) pair – requires the generation of some random numbers.

  Once the (proving key, verification key) pair has been generated from the `.code` file, these random numbers MUST be destroyed. These random numbers MUST never be stored by the party who performed the trusted setup, or that party would be able to generate false proofs which verify as `true`. These random numbers are often referred to as 'toxic waste'.

  Nightfall leverages ZoKrates to perform the trusted setup, and relies on the proper management of the toxic waste by ZoKrates.

  A criticism of zk-SNARKs is that future users of a (proving key, verification key) pair, will have to trust that the party who performed the trusted setup (at the 'beginning of time') did so properly and truthfully.

Figure 5: Security warning: Toxic Waste

Where to look?

`./zkp/code/gm17/nft-mint/nft-mint.pcode`	`.pcode` files with human-readable computations.
`./zkp/code/gm17/nft-mint/nft-transfer.pcode`	
`./zkp/code/gm17/nft-mint/nft-burn.pcode`	
`./zkp/code/gm17/nft-mint/ft-mint.pcode`	
`./zkp/code/gm17/nft-mint/ft-transfer.pcode`	
`./zkp/code/gm17/nft-mint/ft-burn.pcode`	
`./zkp/code/README-tools-trusted-setup.md`	README for automating the trusted setup.
`./zkp/code/README-manual-trusted-setup.md`	README for manually performing the trusted setup.
https://github.com/Zokrates/ZoKrates	ZoKrates source code
https://zokrates.github.io	ZoKrates documentation
`./zkp/code/README-tools-code-preprop.md`	explanation of `.pcode` syntax
https://github.com/EYBlockchain/zokrates-preprocessor	how to manually transpile from `.pcode` to `.code`
`./zkp/code/README-manual-trusted-setup.md`	how to manually do the trusted setup

Deploying the Nightfall Infrastructure

Trusted Benefactor steps:

1. Perform the 'Trusted Setup' to produce the proving key and the verification key for each of the six computations.
2. Share the proving keys with the world (e.g. through an online sharing service). Do not share the proving keys on a blockchain; they're way too big!
3. Locate the Verifier Registry contract's address on the Ethereum mainnet. We intend there to only be one Verifier Registry on the mainnet for all zk-SNARK traffic; in much the same way as the ENS registers all .eth domain names. Note, however, that the default migration scripts of the Nightfall repository do deploy an instance of a Verifier Registry, for example's sake.
4. Either:
   - Locate a GM17 verifier contract address on the Ethereum mainnet; or
   - Deploy an instance of the GM17 verifier contract to the Ethereum mainnet. And register this GM17 verifier contract with the Verifier Registry (see `./zkp/src/vk-controller.js` which does this in the Nightfall repository).
5. Choose which ERC-20 token you wish for your new infrastructure to 'shield'.
6. Deploy an instance of the `FTokenShield.sol` contract to the Ethereum mainnet; specifying the addresses of the chosen GM17 verifier contract and chosen the ERC-20 contract, in the `constructor` of `FTokenShield.sol`.
7. Choose which ERC-721 token you wish for your new infrastructure to 'shield'.
8. Deploy an instance of the `NFTokenShield.sol` contract to the Ethereum mainnet; specifying the addresses of the chosen GM17 verifier contract and chosen the ERC-721 contract, in the `constructor` of `NFTokenShield.sol`.
9. Store all six verification keys in the Verifier Registry. (See `./zkp/src/vk-controller.js` which does this in the Nightfall repository). You will receive six `'vkId'` values from the Verifier Registry in return. These are unique identifiers for the six verification keys.
10. Share the six `'vkId'` values with the world; (e.g. through the same online sharing service as the proving keys). It must be clear to Users which vkId corresponds to which proving key. `./zkp/src/vkIds.json` gives an example of how to store these.
11. Share the Ethereum addresses of the FtokenShield.sol and NFTokenShield.sol contracts.

User steps:

12. Download each proving key and its corresponding vkId from $\mathcal{T}$'s online sharing portal.
13. Generate a `(proof, inputs)` pair, as explained in The Protocols.
14. Submit the `(proof, inputs)` pair to the relevant Shield contract.
    E.g., using web3: `nfTokenShield.mint(proof, inputs, vkId)`
15. Store relevant data in local database.

Figure 4: Deploying the Nightfall Infrastructure

4 Smart Contracts

Contents

4.1 Pre-Existing Contracts . . . 10
- 4.1.1 `ERC-721` . . . 10
- 4.1.2 `ERC-20` . . . 11
- 4.1.3 `ERC-165` . . . 12
- 4.1.4 `Utility contracts` . . . 12

4.2 Shield contracts . . . 12
- 4.2.1 `NFTokenShield.sol` . . . 12
- 4.2.2 `FTokenShield.sol` . . . 13

4.3 Verifier contracts . . . 13
- 4.3.1 `GM17.sol` . . . 13
- 4.3.2 `Points.sol` . . . 13
- 4.3.3 `GM17Library.sol` . . . 13
- 4.3.4 `Pairing.sol` . . . 13

4.4 Verifier Registry contracts . . . 14
- 4.4.1 `Verifier_Registry_Interface.sol` . . . 14
- 4.4.2 `Verifier_Registry.sol` . . . 14
- 4.4.3 `Verifier_Register_Interface.sol` . . . 14

4.5 Deployment of Contracts . . . 14

In this section we give further details of all the Solidity contracts, libraries and interfaces in the Nightfall repository. Some of these contracts are pre-existing and some are new. The important new contracts are:

- Shield contract - stores 'token commitments' which represent ownership of underlying ERC-20 or ERC-721 tokens, and facilitates the minting, transferring and burning of these token commitments.
- Verifier Contract - uses elliptic curve pairing functions to verify a zk-SNARK.
- Verifier Registry Contract - a registry of Verifier Contracts, Verification Keys, and Proof submissions. For simplicity, we ignore this layer from our explanations in this paper; although it is utilised in the Nightfall repository.

Here, we give further details of all Solidity contracts, libraries and interfaces in the Nightfall repository:

4.1 Pre-Existing Contracts

4.1.1 `ERC-721`

The structuring of the ERC-721 contracts is aligned with the https://0xcert.org implementation. These are designed for the transfer of non-fungible assets and include the following files:

`ERC721Interface.sol`	
`ERC721TokenReceiver.sol`	
`ERC721Metadata.sol`	
`NFTokenMetadata.sol`	An example metadata implementation, to accompany `NFToken.sol`
`NFToken.sol`	An example ERC-721 implementation.

An overview of the ERC-721 contracts is given in Figure 6

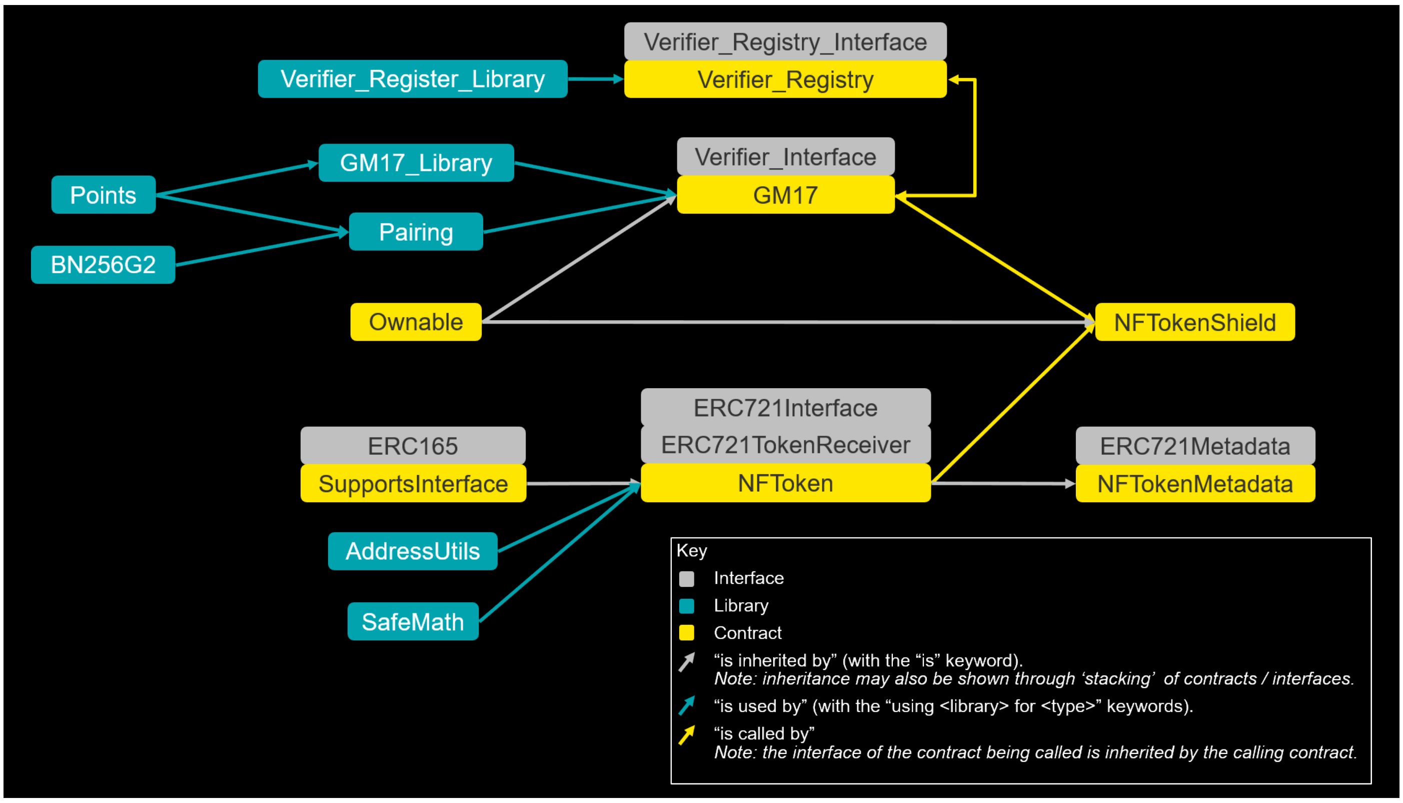

Verifier_Registry_Interface
Verifier_Register_Library
Verifier_Registry
GM17_Library
Verifier_Interface
Points
GM17
Pairing
BN256G2
Ownable
NFTokenShield
ERC721Interface
ERC165
ERC721TokenReceiver
ERC721Metadata
SupportsInterface
NFToken
NFTokenMetadata
AddressUtils
SafeMath
Key
Interface
Library
Contract
"is inherited by" (with the "is" keyword).
Note: inheritance may also be shown through 'stacking' of contracts / interfaces.
"is used by" (with the "using <library> for <type>" keywords).
"is called by"
Note: the interface of the contract being called is inherited by the calling contract.

Figure 6: ERC-721 contract structure.

4.1.2 ERC-20

The structuring of the ERC-20 contracts is aligned with the https://openzeppelin.org implementation. These are designed for the transfer of fungible assets and include the following files:

`ERC20Interface.sol`	
`FToken.sol`	An example ERC-20 implementation.

An overview of the ERC-20 contracts is given in Figure 7

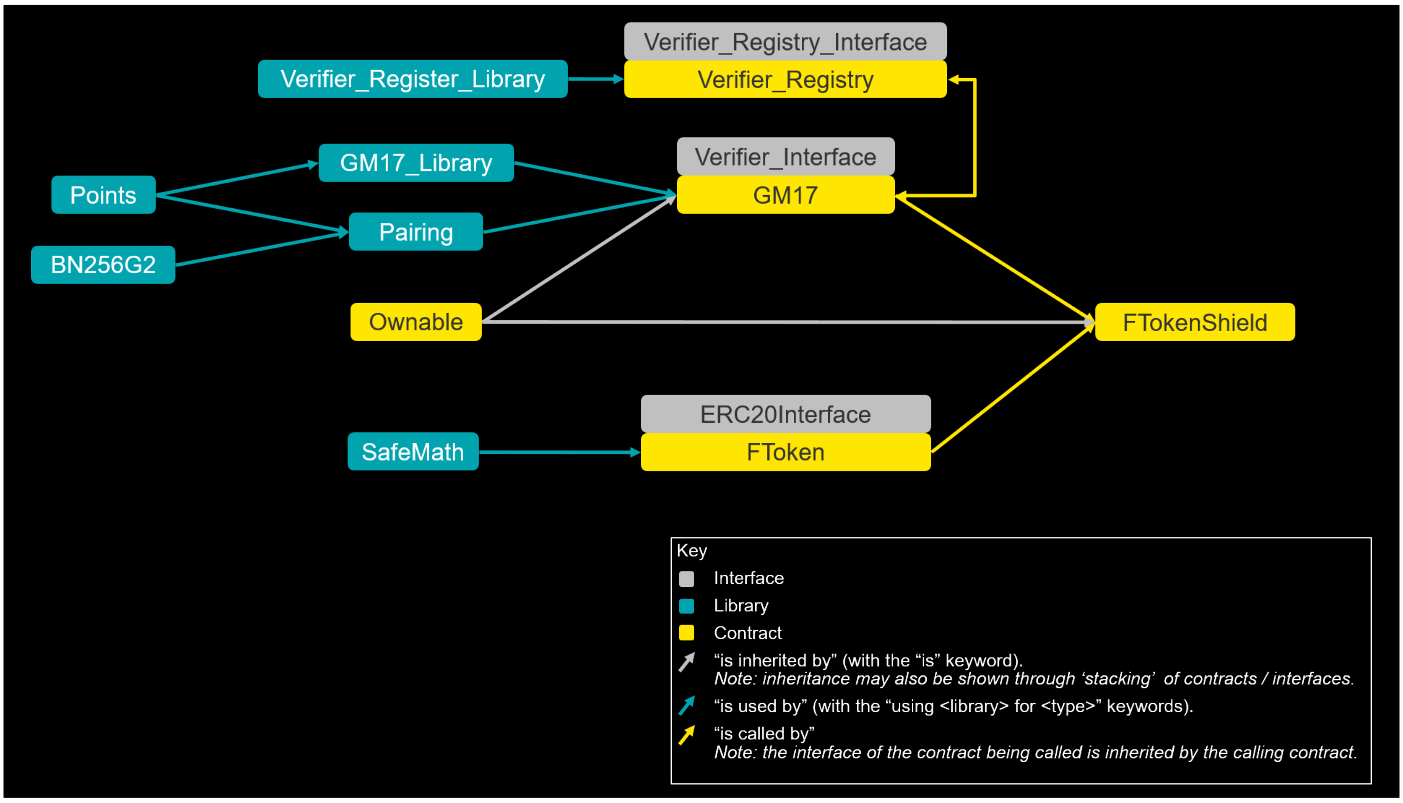

Verifier_Registry_Interface
Verifier_Register_Library
Verifier_Registry
GM17_Library
Verifier_Interface
Points
GM17
Pairing
BN256G2
Ownable
FTokenShield
ERC20Interface
SafeMath
FToken
Key
Interface
Library
Contract
"is inherited by" (with the "is" keyword).
Note: inheritance may also be shown through 'stacking' of contracts / interfaces.
"is used by" (with the "using <library> for <type>" keywords).
"is called by"
Note: the interface of the contract being called is inherited by the calling contract.

Figure 7: ERC-20 contract structure.

4.1.3 `ERC-165`

The structuring of the ERC-165 contracts is aligned with the https://0xcert.org implementation.

`ERC165Interface.sol`
`SupportsInterface.sol`

4.1.4 `Utility contracts`

`AddressUtils.sol`	See https://ethereum.stackexchange.com/a/14016/36603 for more details about how this works.
`SafeMath.sol`	For safe mathematical operations.

4.2 Shield contracts

4.2.1 `NFTokenShield.sol`

Facilitates private transfers of Non-Fungible Tokens.

Constructor: At deployment, specify one Verifier contract (see below) and one ERC-721 contract. NFTokenShield will then be able to hold tokens of the ERC-721 contract in escrow, whilst the private counterparts of these tokens are transferred. Future contributions to Nightfall will produce an NFTokenShield contract which can handle multiple ERC-721 contracts at once.

Stores token commitments, which represent ownership of a token of the specified ERC-721 contract.

Calls upon the Verifier contract to verify zk-SNARKs for it.

A high-level diagram of the 'shielding' process is shown in Figure 8.

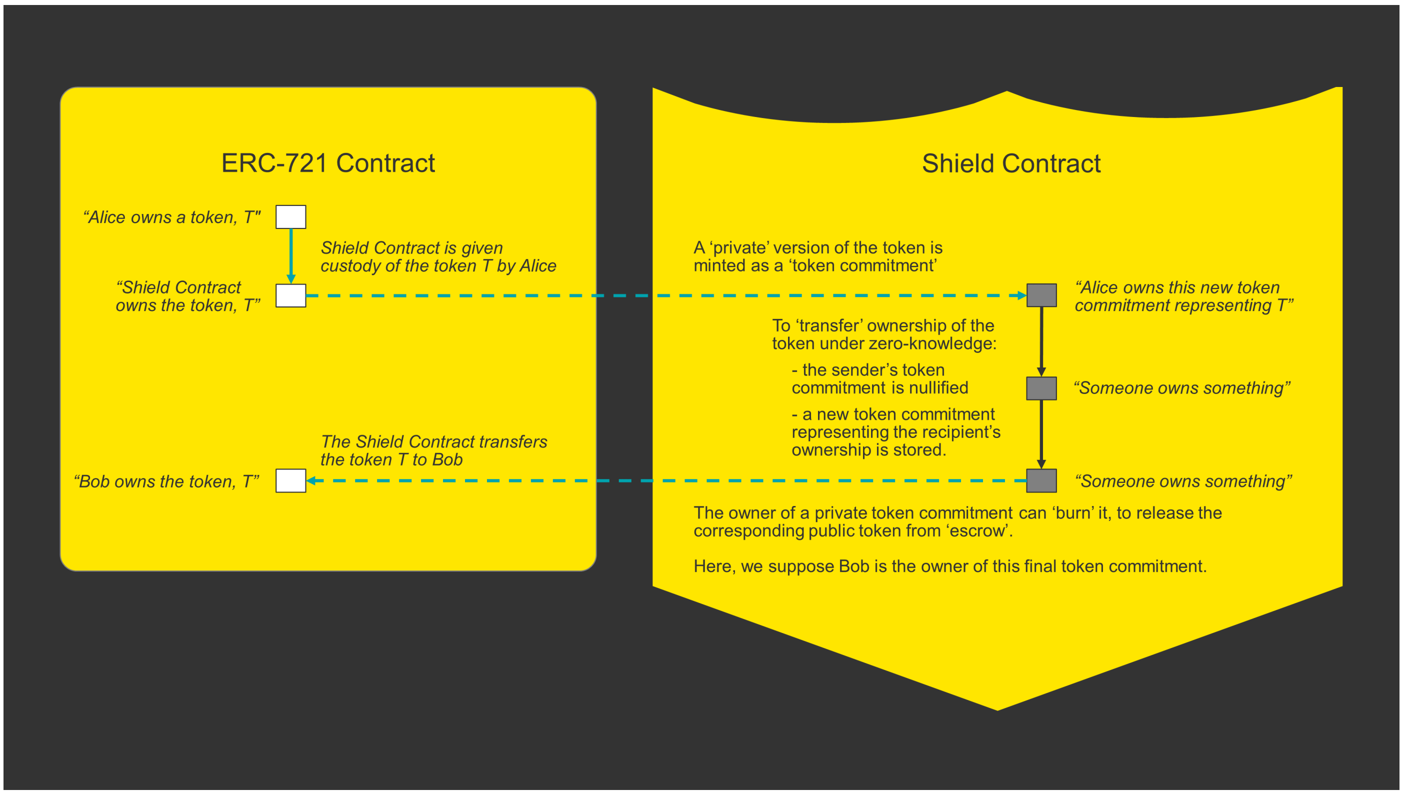

ERC-721 Contract
"Alice owns a token, T"
Shield Contract is given custody of the token T by Alice
"Shield Contract owns the token, T"
The Shield Contract transfers the token T to Bob
"Bob owns the token, T"
Shield Contract
A 'private' version of the token is minted as a 'token commitment'
"Alice owns this new token commitment representing T"
To 'transfer' ownership of the token under zero-knowledge:
- the sender's token commitment is nullified
- a new token commitment representing the recipient's ownership is stored.
"Someone owns something"
"Someone owns something"
The owner of a private token commitment can 'burn' it, to release the corresponding public token from 'escrow'.
Here, we suppose Bob is the owner of this final token commitment.

Figure 8: High-level diagram of 'shielding' an ERC-721 token.

4.2.2 `FTokenShield.sol`

Facilitates private transfers of Fungible Tokens.

Constructor: At deployment, specify one Verifier contract (see below) and one ERC-20 contract. FTokenShield will then be able to hold tokens of the ERC-20 contract in escrow, whilst the private counterparts of these tokens are transferred. Future contributions to Nightfall will produce an FTokenShield contract which can handle multiple ERC-20 contracts at once.

Stores token commitments, which represent ownership of a particular amount of currency, as denominated in the specified ERC-20 contract.

Calls upon the Verifier contract to verify zk-SNARKs for it.

A high-level diagram of the 'shielding' process is shown in Figure 15.

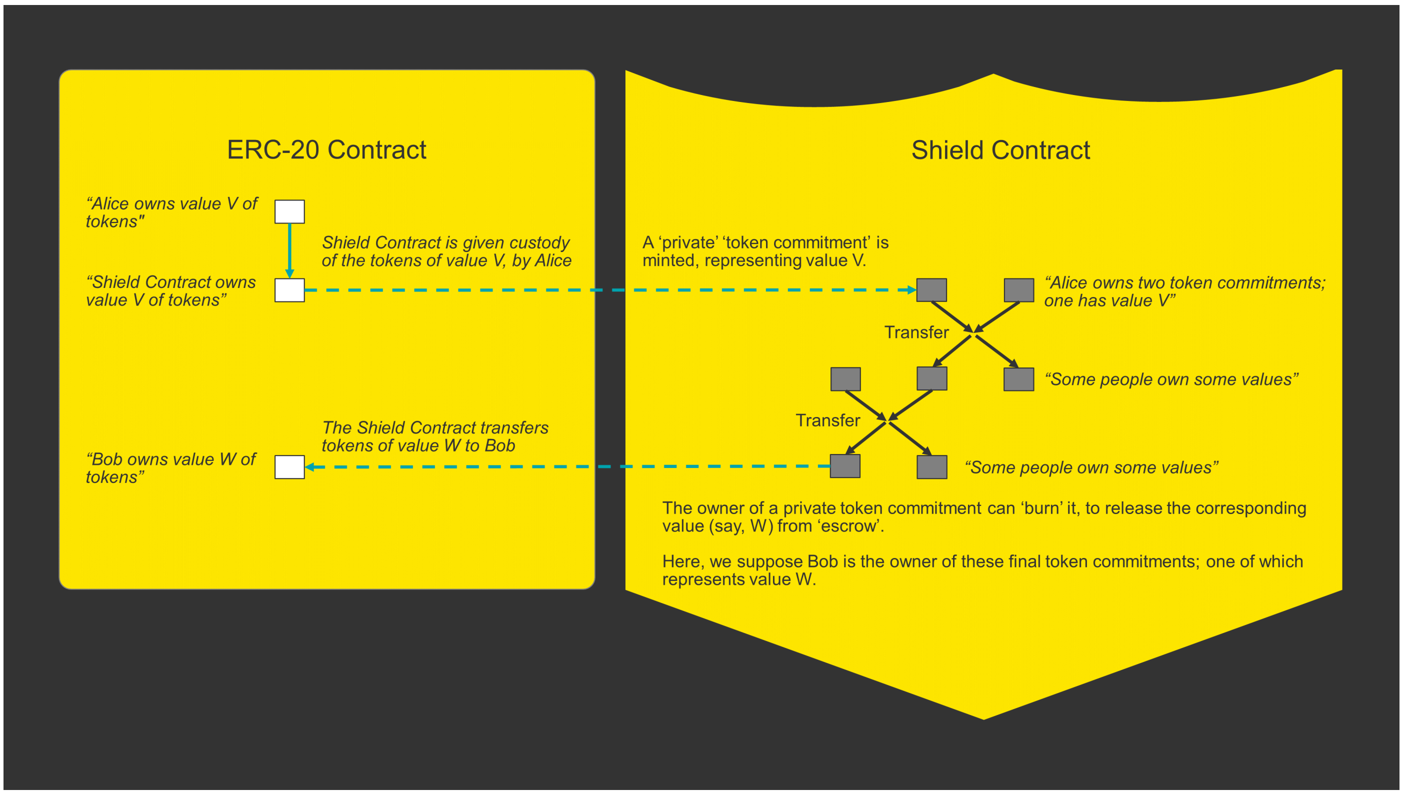

ERC-20 Contract
"Alice owns value V of tokens"
Shield Contract is given custody of the tokens of value V, by Alice
"Shield Contract owns value V of tokens"
The Shield Contract transfers tokens of value W to Bob
"Bob owns value W of tokens"
Shield Contract
A 'private' 'token commitment' is minted, representing value V.
"Alice owns two token commitments; one has value V"
Transfer
"Some people own some values"
Transfer
"Some people own some values"
The owner of a private token commitment can 'burn' it, to release the corresponding value (say, W) from 'escrow'.
Here, we suppose Bob is the owner of these final token commitments; one of which represents value W.

Figure 9: High-level diagram of 'shielding' an ERC-20 token.

4.3 Verifier contracts

The sole purpose of a verifier contract is to verify zk-SNARKs which are passed to it. It returns `true` if the (proof, public inputs) pair verifies. Otherwise, it returns `false`.

4.3.1 `GM17.sol`

An implementation of the draft EIP-1922 zk-SNARK Verifier Standard.

4.3.2 `Points.sol`

Library. Defines how Elliptic Curve coordinates (x, y) are structured.

4.3.3 `GM17Library.sol`

Library. Defines the structures of both a Verification Key and a Proof under the GM17 protocol (using the elliptic curve points of `Points.sol`).

4.3.4 `Pairing.sol`

Library. Performs elliptic curve operations and elliptic curve pairing operations. Utilises the precompiled contracts of EIP-196 and EIP-197.

4.4 Verifier Registry contracts

The Verifier Registry is intended to be a single contract to register all zk-SNARK traffic on the Ethereum mainnet. It facilitates:

– Registration of Verifier contracts

– Storage of all Verification Keys

– Proof submissions

– Other zk-SNARK use-cases beyond Nightfall

Note: although the intention is for there to be just one Verifier Registry on the Ethereum mainnet, the default migration script in the Nightfall repository deploys an implementation of the Verifier Registry along with all other contracts – for the sake of example.

4.4.1 `Verifier_Registry_Interface.sol`

The draft EIP-1923 interface for a Verifier Registry.

4.4.2 `Verifier_Registry.sol`

An implementation of the Verifier_Registry_Interface.

4.4.3 `Verifier_Register_Interface.sol`

Library. Defines the structures of the register, which store entries to the Verifier_Registry.

4.5 Deployment of Contracts

See Trusted Setup for an explanation of deployment steps, and how contract deployment is intertwined with the zk-SNARK trusted setup.

Where to look?

`./zkp/contracts/`	Contracts in Nightfall
`./zkp/migrations/`	Default deployment ordering of contracts in Nightfall
http://eips.ethereum.org/EIPS/eip-165	EIP-165
http://eips.ethereum.org/EIPS/eip-20	EIP-20
http://eips.ethereum.org/EIPS/eip-721	EIP-721
http://eips.ethereum.org/EIPS/eip-196	EIP-196
http://eips.ethereum.org/EIPS/eip-197	EIP-197
http://eips.ethereum.org/EIPS/eip-1922	EIP-1922
http://eips.ethereum.org/EIPS/eip-1923	EIP-1923

5 Microservices

Contents

5.1 zkp **15**
- 5.1.1 `f-token-controller.js` 15
- 5.1.2 `f-token-zkp.js` 15
- 5.1.3 `nf-token-controller.js` 15
- 5.1.4 `nf-token-zkp.js` 15
- 5.1.5 `zokrates.js` 15
- 5.1.6 `vk-controller.js` 15
- 5.1.7 `vkIds.json` 16
- 5.1.8 `stats.json` 16

5.2 offchain **16**
- 5.2.1 whisper 16
- 5.2.2 pkd 16

5.3 accounts **17**

5.4 database **17**

5.5 ui **17**

5.1 zkp

5.1.1 `f-token-controller.js`

Functions to orchestrate mint, transfer, and burn of fungible token commitments.

- Receives public inputs from the front end;
- Calculates the public inputs of each zk-SNARK;
- Calls `zokrates.js` – the ZoKrates JS wrapper – to compute a witness and to generate a proof.
- Calls `f-token-zkp.js` – a web3 transactions module which sends transactions to relevant smart contracts.

5.1.2 `f-token-zkp.js`

Functions to send transactions (relating to fungible commitments) to the smart contracts. Using web3, this js module sends transactions to `FTokenshield`, `GM17`, and `Verifier_Registry`.

5.1.3 `nf-token-controller.js`

Functions to orchestrate mint, transfer, and burn of non-fungible token commitments.

- Receives public inputs from the front end;
- Calculates the public inputs of each zk-SNARK;
- Calls `zokrates.js` – the ZoKrates JS wrapper – to compute a witness and to generate a proof.
- Calls `nf-token-zkp.js` – a web3 transactions module which sends transactions to the `NFTokenShield` contract.

5.1.4 `nf-token-zkp.js`

Functions to send transactions (relating to non-fungible commitments) to the smart contracts. Using web3, this js module sends transactions to `NFTokenshield`, `GM17`, and `Verifier_Registry`.

5.1.5 `zokrates.js`

JS wrapper functions for executing ZoKrates commands within a ZoKrates container. See ZoKrates.

5.1.6 `vk-controller.js`

Functions to send verification keys to the `Verifier_Registry` contract. See Trusted Setup for context.

5.1.7 `vkIds.json`

JSON file which stores the vkId's for each of the six zk-SNARK computations (fungible mint, transfer and burn; and non-fungible mint, transfer and burn).
At the time the verification keys are deployed to the `Verifier_Registry`, it returns a unique vkId for each verification key. `vkIds.json` also stores the Ethereum address of the smart contract to which the verification keys were submitted. See Trusted Setup for context.

5.1.8 `stats.json`

JSON file which stores – for each of the six zk-SNARK computations – the time it took to 'compute-witness' and 'generate-proof' within the ZoKrates container on the User's computer. These time statistics serve as 'ETA' estimates for the next time the User generates a proof (and the command line displays a progress bar accordingly).

Where to look?

`./zkp/`	The zkp microservice

5.2 offchain

5.2.1 whisper

Where to look?

https://github.com/ethereum/wiki/wiki/Whisper	Whisper GitHub.
https://web3js.readthedocs.io/en/1.0/web3-shh.html	web3 for whisper
`./offchain/whisper-controller-stub.js`	A whisper js wrapper for Ganache
`./offchain/whisper-controller.js`	A whisper js wrapper for Geth
`./offchain/listners.js`	Listens for messages on behalf of the user. Decrypts relevant messages. Forwards the data to the relevant microservice to take action. E.g. to store new data in the database.

LIMITATION

The `whisper-controller.js` and The `whisper-controller-stub.js` only listen for Whisper events (via the '`subscribe`' methods) **when the user is logged into the Application**.

If a user logs out, they will miss any incoming Whisper messages. E.g. Bob might not receive notification from Alice that he has been sent a commitment, and will not receive details of the preimage of the commitment, nor the location of the commitment within the on-chain Merkle Tree.

This can be solved with future contributions to the Nightfall repository. Indeed, web3.shh includes the functionality to retrieve past messages already.

Figure 10: Limitation: Nightfall does not currently receive Whisper messages if the User is not logged in.

5.2.2 pkd

Nightfall uses a PKD (Public Key Directory) contract to allow users to lookup both ZKP public keys and Whisper public keys. See The Protocols for a disambiguation of the different public keys used in Nightfall. The public keys of a user can be retrieved with knowledge of their Ethereum Address.

The PKD also serves as a simple Name Service; users can register a unique name with the PKD. With this, the public keys of a user can also be retrieved with knowledge of their unique name.

An example usage of the PKD is:
If Bob wishes to ask Alice to send him an ERC-721 commitment under zero-knowledge: Bob can query Alice's Whisper public key from the PKD, and Alice can then query Bob's ZKP public key from the PKD.

5.3 accounts

The ‘accounts’ microservice manages a User’s Ethereum accounts. (We use the terms Ethereum ‘address’ and Ethereum ‘account’ interchangeably).

For a user, Alice, her anonymity is preserved by using a new ‘throwaway’ Ethereum address each time she transacts with the Shield contract.

This microservice generates new Ethereum accounts, and keeps track of them for the application.

“But how would Alice pay for the gas costs of sending such a transaction to the Shield contract?”

She would have to pay for the verification computation of her zk-SNARK, and for the persistent storage of the public inputs to her zk-SNARK. In order for Alice to fund a new Ethereum account completely anonymously, she would have to mine Ether. This might not be a viable solution for some; as mining rewards can be unpredictable and could be insufficient to cover the gas needed to transact using Nightfall.

Alternatively, Alice could make each transaction through a delegated third-party, who would send the `‘transfer’` transaction on Alice’s behalf. The initial release of Nightfall does not include functionality to delegate transactions to others. Nevertheless, we know that in future updates we can solve the problem of hiding that “Alice transferred something” so that observers only see that “someone transferred something”.

PRIVACY WARNING

The initial release of Nightfall does not give Alice full anonymity when she interacts with the Shield contract, unless she mines into her anonymous Ethereum accounts.

Future updates will include the functionality to delegate transactions to others. This is a solved problem, which just needs to be implemented.

Figure 11: Privacy warning: A future update is required to Nightfall to allow user’s to reliably and consistently transact with the Shield contract anonymously.

Where to look?

`./accounts/` The accounts microservice

5.4 database

Nightfall uses mongodb to store private data on a User’s local machine.

SECURITY WARNING

Currently, the ‘secret keys’ for spending token commitments are stored in a User’s ‘User’ db. This is not particularly secure, and moderations might need to be made when creating production-ready applications.

Figure 12: Security warning: Secret keys are currently stored in the User’s db.

Where to look?

`./database/src/models/` All schemas.

5.5 ui

See the dedicated README for instructions on how to use the UI.

SECURITY WARNING

Currently, random salt values (denoted σ in this document) are generated within the UI microservice, or within the api-gateway microservice.

Ensure you're comfortable with the level of randomness achieved by these random number generators.

Figure 13: Security warning: Ensure you're comfortable with any random number generation in the application

Where to look?

`./ui-src/`	The UI microservice
`UI.md`	A demonstration of the UI

6 Finite Fields and Bit Lengths

For those who go through the Nightfall code, you mght realise there are many number conversions being made in the zkp microservice. In particular, there are frequently conversions and restrictions to 216-bit (27-byte) values in much of today's Nightfall code. These conversions are a consequence of working with zk-SNARKs.

Like many cryptographic protocols, zk-SNARKs make use of representing numbers as points on an elliptic curve. In doing so, the results of our computations become quite difficult to 'unravel' – that is, given an output, it becomes computationally infeasible for someone to determine the inputs. However, it also restricts the mathematics we can do. This is a gross oversimplification of why elliptic curves are used, but let's talk about them.

Currently on Ethereum, there is only one elliptic curve for which it is 'cheap' (in terms of gas costs) to perform calculations (due to there being precompile contracts supporting calculations on this curve):

$$E := y^2 = x^3 + 3$$

This curve looks like this:

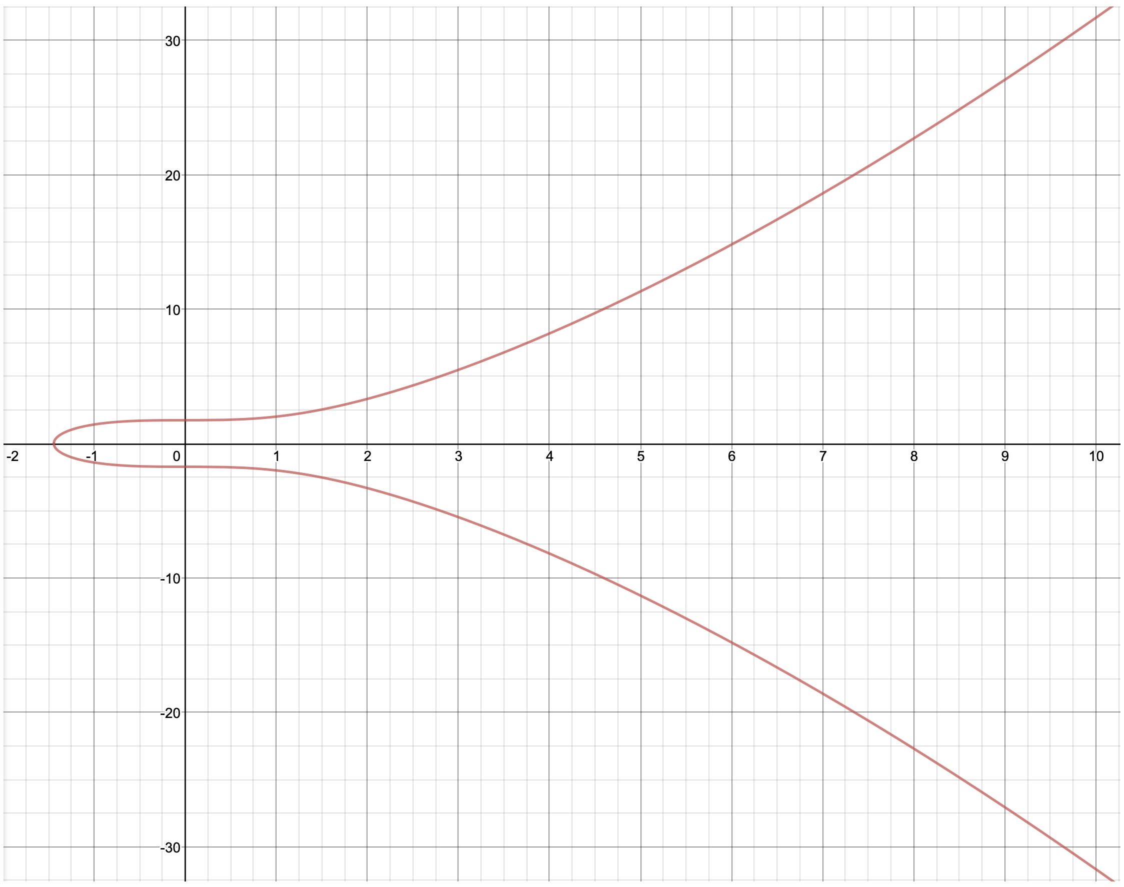


Figure 14: $y^2 = x^3 + 3$ over the real numbers.

As an example, let's restrict the x and y coordinates to be the field of integers modulo 7. I.e. we only allow the numbers $\mathbb{F}_7 = 0, 1, 2, 3, 4, 5, 6$. In this world, $5 + 5 = 10 = 3\ (mod\ 7)$.

Let's consider the possible y-values of our elliptic curve E when restricted to $\mathbb{F}_7$ (note: we write $E[\mathbb{F}_7]$ for 'the elliptic curve defined over the finite field $\mathbb{F}_7$).

y	0	1	2	3	4	5	6
y^2	0	1	4	$9 = 2$	$16 = 2$	$25 = 4$	$36 = 1$

Now let's consider the x-values of $E[\mathbb{F}_7]$:

x	0	1	2	3	4	5	6
x^3	0	1	$8 = 1$	$27 = 6$	$64 = 1$	$125 = 6$	$216 = 6$
$x^3 + 3$	3	4	4	$9 = 2$	4	$9 = 2$	$9 = 2$
$= y^2$	-	4	4	2	4	2	2
Valid y-values	–	2 5	2 5	3 4	2 5	3 4	3 4

So we have a set of valid points of $\infty, (1,2), (1,5), (2,2), (2,5), (3,3), (3,4), (4,2), (4,5), (5,3), (5,4), (6,3), (6,4)$.

These 13 points are the only points which exist on $E[\mathbb{F}_7]$. $E[\mathbb{F}_7]$ is a 'group' of order 13.
In other words: the curve $E : y^2 = x^3 + 3$ — when restricted to the 7 values of $\mathbb{F}_7$ — produces a group $E[\mathbb{F}_7]$ of order 13.

We superimpose the points of $E[\mathbb{F}_7]$ (in green) below:

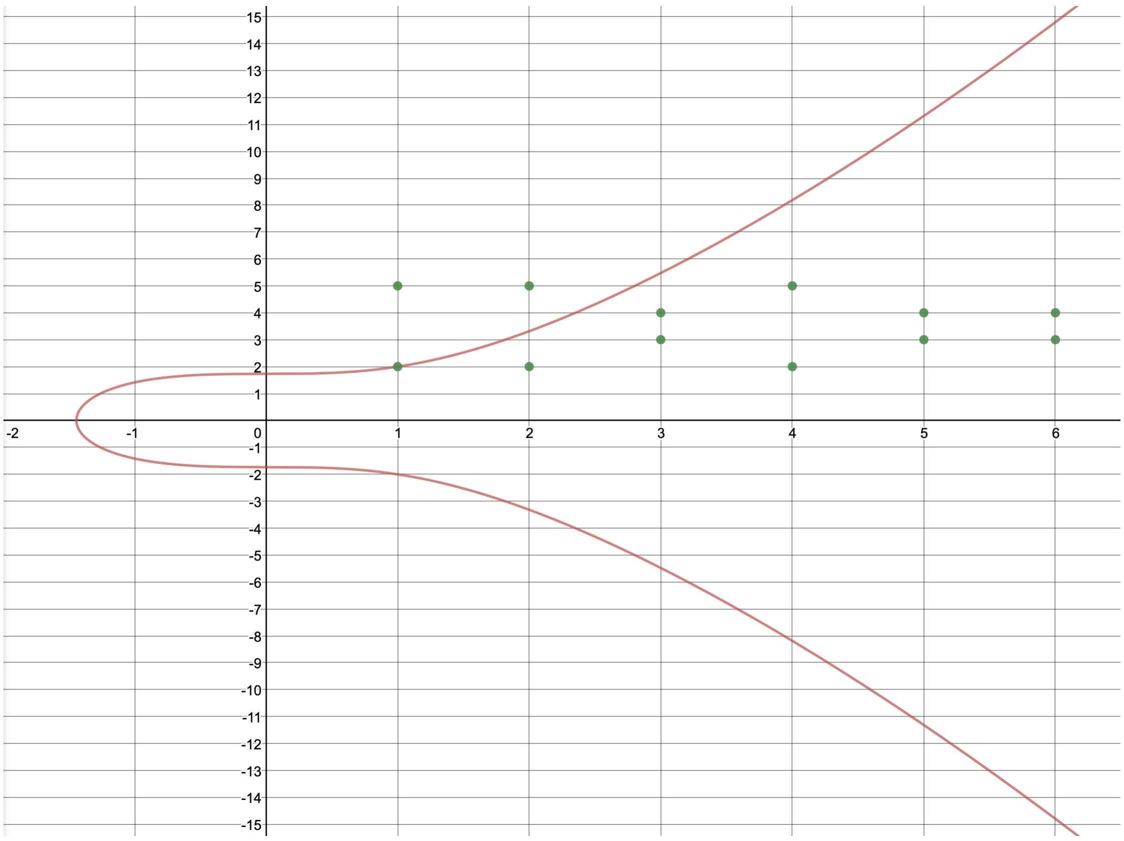


Figure 15: $y^2 = x^3 + 3$ over $\mathbb{F}_7$ shown as green dots.

An important thing to take away from this example, is that there are 3 distinct things for us to be aware of: an elliptic curve equation E, a finite field $\mathbb{F}$, and the resulting group $G = E[\mathbb{F}]$. Sometimes the number of elements in the group G is *more than* the number of elements in the field $\mathbb{F}$ (as in the above example), and sometimes the number of elements in the group G is *less than* the number of elements in the field $\mathbb{F}$ (as we will see is the case with Ethereum).

In practice, the finite field $\mathbb{F}_p$ used in Ethereum is of size

$$p = 21888242871839275222246405745257275088696311157297823662689037894645226208583$$

The elliptic curve $E : y^2 = x^3 + 3$, when restricted to the p values of $\mathbb{F}_p$, produces a group $G_1 = E[\mathbb{F}_p]$ of prime order

$$q = 21888242871839275222246405745257275088548364400416034343698204186575808495617$$

I.e. $G_1 = E[\mathbb{F}_p]$ is a group with q distinct points. Here, $q < p$; the size of the group G_1 is less than the size of the field $\mathbb{F}_{|}$.

This has an important consequence when working within ZoKrates. To generate zk-SNARKs, ZoKrates will convert the numbers we pass as inputs to our off-chain calculation (as well as all intermediate numbers of the calculation) into elliptic curve points. To tie-in with Ethereum, ZoKrates converts all of the numbers used in its calculations into elements of the group G_1. Therefore, we need to make sure that **all** of the inputs we pass into ZoKrates are **less than** the size of the group G_1, $q = 21888242871839275222246405745257275088548364400416034343698204186575808495617$.

This value of q is slightly less than 254-bits. I.e. $2^{253} < q < 2^{254}$.

Hence, to be sure we don't 'overflow' modulo-q, it's safest to only pass numbers to ZoKrates which are $\leq$ 253-bits. When working with Ethereum, we often work with hex numbers (because Solidity 'likes' hex numbers), which means it's often nice for the bit-length we work with to be divisible by 8. Since 248 is the largest number below 253 which is divisible by 8, this would have been a nice choice for all of the numbers Nightfall passes to ZoKrates.

However, you might notice we instead restrict all our numbers to 216-bits. The reason for this is the maximum size of a message in the NIST specification of the sha256 hashing algorithm. The largest message size for ‘one round’ of sha256 hashing is 447-bits. The version of ZoKrates used in Nightfall currently only supports ‘one round’ of sha256 hashing. Hence our inputs to each hashing iteration must be 447-bits or less.

You’ll see in the protocols below, that we frequently need to concatenate two values and then hash them. Hence we need each of the two values to be at most 223-bits to fit inside ‘one round’ of sha256 hashing. Given that we also prefer bit-lengths which are divisible by 8, 216-bits becomes the best choice for our purposes.

And that’s why you’ll see conversions and restrictions to 216-bit (27-byte) values in much of today’s Nightfall code.

In cases where we need to use a number with a greater bit-length than 216 (for security purposes), we deconstruct that number into an array of 216-bit values. E.g. a 512-bit number might be deconstructed into an array [80-bits, 216-bits, 216-bits].

Part III

The Protocols

Table of Contents

7 ERC-721 (non-fungible) tokens **23**
- 7.1 Mint . . . 27
- 7.2 Transfer . . . 31
- 7.3 Burn . . . 38

8 ERC-20 (fungible) Tokens **44**
- 8.1 Mint . . . 47
- 8.2 Transfer . . . 51
- 8.3 Burn . . . 59

7 ERC-721 (non-fungible) tokens

Contents

7.0.1 Variables 25
7.0.2 Key Management 26
7.1 Mint 27
7.1.1 Details 29
7.2 Transfer 31
7.2.1 Details 34
7.3 Burn 38
7.3.1 Details 41

In this section we give an overview of using Nightfall to privately transact non-fungible tokens (nft's). We cover three key functions:

- Mint - create an initial 'token commitment'; a private representation of a public ERC-721 token.
- Transfer - nullify the sender's token commitment, and generate a new token commitment to represent ownership by the recipient.
- Burn - nullify a token commitment, and receive the underlying public ERC-721 token.

7.0.1 Variables

Here we describe the variables used in this section.

A, B	Participants Alice and Bob.
pk_A	The public key belonging to Alice.
sk_A	The secret key belonging to Alice. Note: there are several (secret key, public key) pairs in this protocol)
E_A	The Ethereum address of Alice.
$\Xi_{A,i}$	An 'anonymous' Ethereum address belonging to Alice, where $i \in \mathbb{N}$ is an index, for distinguishing between multiple anonymous addresses.
α	A unique representation of some non-fungible asset e.g. a tokenId in ERC-721. Note that in respect of non-fungible tokens, Nightfall currently focusses solely ERC-721 tokens. It would be relatively simple to adapt Nightfall's application to deal with other non-fungible token standards.
α_A	A non-fungible asset α that is in Alice's possession.
σ	A salt used to provide uniqueness to commitment preimages.
$\sigma_{\vec{AB}}$	Stresses that a salt is being shared privately from Alice to Bob.
Z	An ERC-721 commitment; a zero-knowledge commitment representing ownership of some underlying ERC-721 asset.
Z_A	Stresses that an ERC-721 commitment belongs to Alice.
Z_α	Stresses that an ERC-721 commitment represents the asset α.
Z_l	Stresses that an ERC-721 commitment is the l^{th} leaf of a Merkle Tree (see below for M). Note that the meaning of these (seemingly colliding or ambiguous) subscripts will be clear from context.
N	A nullifier for an ERC-721 commitment Z.
N_A	A nullifier for the ERC-721 commitment Z_A.
N_α	A nullifier for the ERC-721 commitment Z_α.
M	A binary Merkle Tree.
M_l	A binary Merkle Tree with l non-zero leaves (where leaves are populated in order 'from left to right').
root_l	The root of M_l ('M' is omitted because context will be clear).
ϕ_L	$[\phi_L(d-1), \phi_L(d-2), ..., \phi_L(1), \phi_L(0)]$ - The path from a leaf L to the root of a Merkle Tree M, where $\phi_L(0) = \mathsf{root}$.
ϕ	$[\phi_{d-1}, \phi_{d-2}, ..., \phi_1, \phi_0]$ - Alternative notation for the path from a leaf, where the leaf L is clear from the context. $\phi_0 = \mathsf{root}$.
ψ_L	$[\psi_L(d-1), \psi_L(d-2), ..., \psi_L(1), \psi_L(0)]$ - The sister-path from a leaf L to the root of a Merkle Tree M, where $\psi_L(0) = \phi_L(0) = \mathsf{root}$.
ψ	$[\psi_{d-1}, \psi_{d-2}, ..., \psi_1, \psi_0]$ - Alternative notation for the sister-path from a leaf, where the leaf L is clear from the context. $\psi_0 = \mathsf{root}$.
x	Public inputs to a zk-SNARK.
ω	Private inputs to a zk-SNARK.
C	An arithmetic circuit $C : (\omega, x) \to \{0, 1\}$.
p_C	A proving key for the circuit C. (Not to be confused with pk which denotes a public key).
vk_C	A verification key for the circuit C.
$\pi(p_C, x, \omega)$	A proof for the circuit C, public inputs x, and private inputs ω
$\pi_{C,x,\omega}$	An abbreviation of the above.
π	An abbreviation of the above, when the context of the proof is clear.
$h()$	A one-way hashing function. Nightfall currently uses sha256 hashing throughout.

7.0.2 Key Management

There are several pairs of public and private keys to keep track of throughout these protocols. We provide a summary here (for an actor Alice (A)):

Ownership of...	**Account symbol**	**Private Key**	**Public Key**	**Notes**
Ethereum address	E_A	sk_A^E	pk_A^E	Used for 'mint' and 'burn'.
Anonymous Ethereum addresses	$\Xi_{A,i}$	$sk_{A,i}^{\Xi}$	$pk_{A,i}^{\Xi}$	Used for 'transfer'. $i \in \mathbb{N}$.
Ethereum Whisper accounts	$W_{A,j}$	$sk_{A,j}^W$	$pk_{A,j}^W$	Used for private messaging. $j \in \mathbb{N}$
ERC-721 commitment Z_l	Z_l	$sk_A^{Z,l}$	$pk_A^{Z,l}$	Used to 'mint', 'transfer' and 'burn' Z_l.

Hereafter, when we write sk_A and pk_A we will be referring to $sk_A^{Z,l}$ and $pk_A^{Z,l}$ respectively (where Z_l is clear from context) - unless otherwise stated.

7.1 Mint

Suppose Alice wishes to be able to transfer ownership of an ERC-721 token under zero-knowledge, so that the following become private:

1. All details of the ERC-721 token (the 'asset').
2. The identity of the sender of the token ('Alice').
3. The identity of the recipient of the token.

In order to achieve this, Alice must first convert her ERC-721 token into a private ERC-721 commitment. We call this act of conversion **'minting'** an ERC-721 commitment. In this section, we outline Nightfall's protocol for minting an ERC-721 commitment, but first, an important privacy warning:

PRIVACY WARNING

Privacy is NOT achieved during the minting stage!

Minting an ERC-721 commitment initially requires Alice to transfer her ERC-721 token to a 'Shield' contract (which thereafter holds it in escrow). This transfer reveals the Ethereum address of the sender (Alice) as well as the ERC-721 token itself. Therefore everyone will know the owner and the underlying asset being represented by the initial ERC-721 commitment which is created at this 'minting' stage.

Only during subsequent 'transfers' of the new ERC-721 commitment, will we achieve the privacy intentions of Figure **??**

The ERC-721 standard allows many unique assets to be tokenised and represented by a unique tokenId within an ERC-721 smart contract. Let α be the tokenId of some ERC-721 asset.

For Alice to mint a token commitment representing α, on the blockchain, under zero knowledge, she follows the steps in Figure 16:

Non-fungible mint algorithm

Alice's steps:

1. Generate a random salt σ_A.
2. Compute $Z_A := h(\ \alpha \mid pk_A^Z \mid \sigma_A\ )$, a token commitment which represents α.
3. Set public inputs $x = (\ \alpha,\ Z_A\ )$
4. Set private inputs $\omega = (\ pk_A^Z,\ \sigma_A\ )$
5. Select $C_{nft\text{-}mint}(\ \omega,\ x\ )$ – the set of constraints which are satisfied if and only if:
    (a) Z_A equals $h(\ \alpha \mid pk_A^Z \mid \sigma_A\ )$ (Proof that the commitment Z_A hides the correct asset α)
6. Generate $\pi := P(\ p_C\ ,\ x,\ \omega\ )$; a proof of knowledge of satisfying arguments (ω, x) $s.t.$ $C(\omega, x) = 1$. Recall: p_C – the proving key for C – will be stored on Alice's computer.

   The pair (π, x) is the zk-SNARK which attests to knowledge of private inputs ω without revealing them.
7. Send (π, x) to the Shield contract for verification.

   Using web3: `nfTokenShield.mint(proof, inputs, vkId)`

Shield contract's steps:

8. Verify the proof as correct: call a Verifier contract to verify the `(proof, inputs)` pair against the verification key represented by `vkId`.

Verifier contract's steps:

9. Compute `result = verify(proof, inputs, vkId)`.

   I.e. Verify the `(proof, inputs)` pair against the verification key.
10. Return `result`$\in\{$`false, true`$\}$ to the Shield contract.

Shield contract's steps:

11. If `result = false`, revert.
12. Else:
    (a) Transfer α (the ERC-721 token with `tokenId` $= \alpha$), on behalf of Alice, to the Shield Contract. I.e. store α in escrow.
    (b) Add Z_A to the next empty leaf of the Merkle Tree.
    (c) Recalculate the path to the root of the Merkle Tree from Z_A for future users.

Alice's steps:

13. Store relevant data in local database, including the leafindex of Z_A.

Figure 16: Non-Fungible Mint Algorithm

7.1.1 Details

We refer to the numbered steps of Figure 16.

Step 1
This is handled within the UI microservice (or within the api-gateway).

Steps $2 - 4$
These steps are handled within `nf-token-controller.js`.

Steps $5 - 6$
These steps are handled within a ZoKrates container.

Step 7
This transaction is handled within `nf-token-zkp.js`.

Steps $8 - 10$
The Verifier contract already has stored within it the object vk_C (see Trusted Setup). It runs a verification function $V(vk_C, \pi, x)$.

$$V : (vk_C, \pi_{C,x,\omega}, x) \to \{0, 1\}$$

where:

$$V = \begin{cases} 1, & \text{if } \pi_{C,x,\omega} \text{ and } x \text{ satisfy } vk_C \\ 0, & \text{otherwise} \end{cases}$$

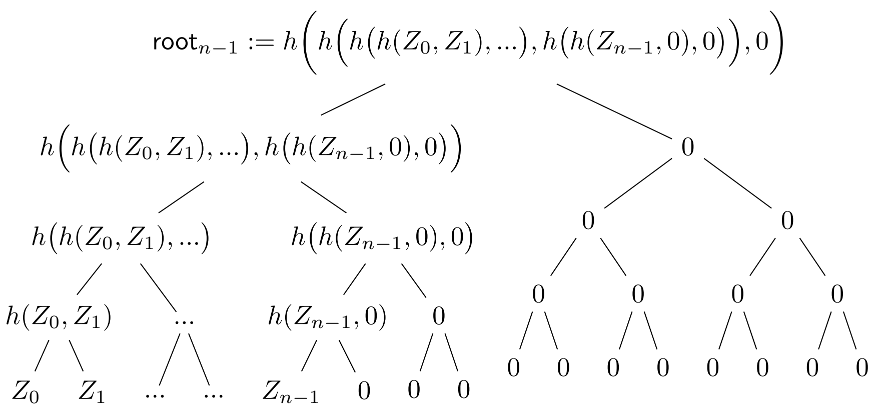
Steps $11 - 12$
If the Verifier contract returns 1 (`true`) (verified) to the Shield contract, then the Shield contract will be satisfied with Alice's commitment, and will update its persistent states:

Suppose the Shield contract stores an ever-increasing array, $\boldsymbol{Z}$, of all token commitments which have ever been submitted by anyone.

Suppose, prior to Alice's mint, there are $n - 1$ tokens in the tree:

$$\boldsymbol{Z}_{n-1} = (Z_0, Z_1, ..., Z_{n-1})$$

The information held within $\boldsymbol{Z}_{n-1}$ may be represented by the root hash root_{n-1} of a Merkle Tree M_{n-1}:

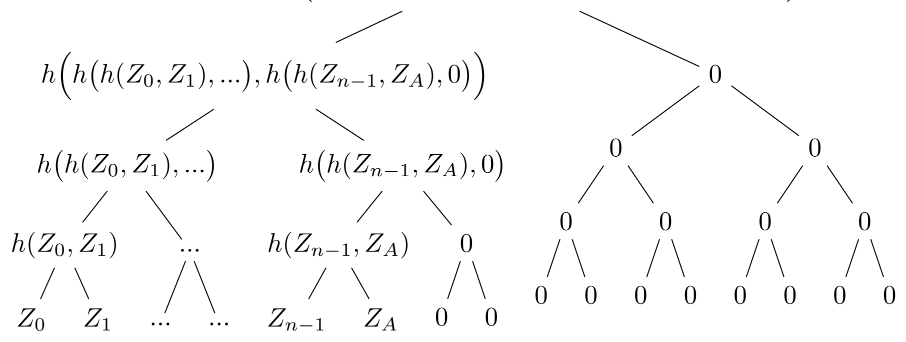

$\mathsf{root}_{n-1} := h\Big(h\Big(h(h(Z_0, Z_1), ...), h(h(Z_{n-1}, 0), 0)\Big), 0\Big)$
$h\Big(h(h(Z_0, Z_1), ...), h(h(Z_{n-1}, 0), 0)\Big)$
0
$h(h(Z_0, Z_1), ...)$
$h(h(Z_{n-1}, 0), 0)$
0
0
$h(Z_0, Z_1)$
...
$h(Z_{n-1}, 0)$
0
0 0 0 0
Z_0 Z_1 ... ...
Z_{n-1} 0 0 0
0 0 0 0 0 0 0 0

Now that the Shield contract has been given verification that Alice's commitment, Z_A, does indeed hide the asset α, the Shield contract will do the following:

- Append the commitment Z_A to the ever-increasing array of tokens, $\boldsymbol{Z}_{n-1}$, so that $\boldsymbol{Z}_n = (Z_0, Z_1, ...Z_{n-1}, Z_A)$

- Recalculate a Merkle Root root_n of M_n:

$$\mathsf{root}_n := h\Big(h\Big(h\big(h(Z_0, Z_1), ...\big), h\big(h(Z_{n-1}, Z_A), 0\big)\Big), 0\Big)$$


$h\Big(h\big(h(Z_0, Z_1), ...\big), h\big(h(Z_{n-1}, Z_A), 0\big)\Big)$
0
$h\big(h(Z_0, Z_1), ...\big)$
$h\big(h(Z_{n-1}, Z_A), 0\big)$
0
0
$h(Z_0, Z_1)$
...
$h(Z_{n-1}, Z_A)$
0
0
0
0
0
Z_0
Z_1
...
...
Z_{n-1}
Z_A
0
0
0
0
0
0
0
0
0
0

- Append root_n to an ever-increasing array $\textbf{roots} = (\mathsf{root}_0, \mathsf{root}_1, ..., \mathsf{root}_{n-1}, \mathsf{root}_n)$

Step 13
Alice will store all important information in her private database.

7.2 Transfer

We continue with the notation and indices from the 'Mint' section.

Suppose Alice wishes to transfer ownership of the ERC-721 token with tokenId 'α' to Bob, but under zero-knowledge.

In the 'Mint' section, we saw how Alice can create an 'ERC-721 commitment' Z_α within the Shield contract which:

- hides an underlying ERC-721 token with tokenId 'α'; and
- hides and binds Alice as the owner of Z_α (and hence of α) through an ownership keypair (sk_A^Z, pk_A^Z).

Recall our privacy intentions:
Alice wishes to be able to transfer ownership of an ERC-721 token under zero-knowledge, so that the following become private:

1. All details of the ERC-721 token (the 'asset').
2. The identity of the sender of the token ('Alice').
3. The identity of the recipient of the token.

Recall that minting a token commitment does not yet afford Alice any privacy (see the warning in Figure **??**). Only with subsequent transfers will the whereabouts of α and the owner of α be hidden.

For Alice to transfer ownership of α within the Shield contract, under zero knowledge, she follows the steps in Figure 17.

Non-fungible transfer algorithm

Bob's steps:

1. Before Alice can send him anything, Bob must register his public keys pk_B^Z and pk_B^W against both his public Ethereum address pk_B^E and his unique name 'Bob' within the PKD.

Alice's steps:

2. Generate a random salt $\sigma_{\vec{AB}}$.
3. Lookup Bob's 'zkp' public key pk_B^Z from the PKD.
4. Compute $Z_B := h(\ \alpha \mid pk_B^Z \mid \sigma_{\vec{AB}}\ )$, a token commitment which represents α.
5. Compute $N_A := h(\ \sigma_A \mid sk_A^Z\ )$, the nullifier of Alice's commitment Z_A.
6. Get ψ_{Z_A} – the sister-path of Z_A – from the Shield contract (see Details below).
7. Get the latest Merkle root from the Shield contract: $\texttt{root}_{n+m-1}$ (see Details below).
8. Set public inputs $x = (\ N_A,\ \texttt{root}_{n+m-1},\ Z_B)$
9. Set private inputs $\omega = (\alpha,\ \psi_{Z_A},\ sk_A,\ \sigma_A,\ pk_B,\ \sigma_{\vec{AB}})$
10. Select $C_{nft-transfer}(\ \omega,\ x\ )$ – the set of constraints which are satisfied if and only if:
    (a) pk_A equals $h(\ sk_A\ )$; (Proof of knowledge of the secret key to pk_A) (see Details for why pk_A isn't an input to C)
    (b) Z_A equals $h(\ \alpha \mid pk_A \mid \sigma_A\ )$ (Proof of the constituent values of Z_A) (see Details for why Z_A isn't an input to C)
    (c) $\texttt{root}_{n+m-1}$ equals $h\Big(\psi_1 \mid ... \mid h\Big(\psi_{d-2} \mid h\big(\psi_{d-1} \mid Z_A\big)\Big)...\Big)$ (Proof that Z_A belongs to the on-chain Merkle Tree)
    (d) N_A equals $h(\ \sigma_A \mid sk_A^Z\ )$ (Proof N_A is indeed the nullifier of Z_A)
    (e) Z_B equals $h(\ \alpha \mid pk_B^Z \mid \sigma_{\vec{AB}}\ )$ (Proof that Z_B contains the same asset as Z_A)
11. Generate $\pi := P(\ p_C\ ,\ x,\ \omega\ )$; a proof of knowledge of satisfying arguments (ω, x) *s.t.* $C(\omega, x) = 1$. Recall: p_C – the proving key for C – will be stored on Alice's computer.

    The pair (π, x) is the zk-SNARK which attests to knowledge of private inputs ω without revealing them.
12. Send (π, x) to the Shield contract for verification.

    Using web3: `nfTokenShield.transfer(proof, inputs, vkId)`

Shield contract's steps:

13. Verify the proof as correct: call a Verifier contract to verify the `(proof, inputs)` pair against the verification key represented by `vkId`.

...

Figure 17a: Non-Fungible Transfer Algorithm

Verifier contract's steps:

14. Compute `result = verify(proof, inputs, vkId)`.
    I.e. Verify the `(proof, inputs)` pair against the verification key.
15. Return `result`$\in${`false, true`} to the Shield contract.

Shield contract's steps:

16. If `result = false`, revert.
17. Else:
    (a) Check root_{n+m-1} is in **roots**. (Revert if not).
    (b) Check N_A is not already in its list of 'spent' nullifiers. (Revert if not).
    (c) Add Z_B to the next empty leaf of the Merkle Tree.
    (d) Recalculate the path to the root of the Merkle Tree from Z_B for future users.
    (e) Append the newly calculated root root_{n+m} to the ever-increasing array **roots**
    (f) Similarly append the nullifier N_A to the ever-increasing array $\boldsymbol{N}$.

Alice's steps:

18. Store relevant data in her local database, including the leafindex of Z_B.
19. Send Bob important data privately via Whisper (using his public key pk_B^W):
    (a) The salt $\sigma_{\vec{AB}}$ of Z_B.
    (b) The public key of Bob, pk_B^Z, used by Alice in the preimage of Z_B (for completeness, so Bob can check the correctness of Z_B himself).
    (c) The tokenId α.
    (d) Z_B.
    (e) The leafIndex of Z_B within the on-chain Merkle Tree M (so Bob can locate it).

Bob's steps:

20. Check the correctness of the information provided by Alice:
    (a) Check Z_B equals $h(\ \alpha \mid pk_B^Z \mid \sigma_{\vec{AB}}\ )$
    (b) Check that Z_B is stored at the leafIndex of M which Alice claimed.
21. Store relevant data in his local database, including whether or not his 'correctness checks' passed.

Figure 17b: Non-Fungible Transfer Algorithm

7.2.1 Details

We refer to the numbered steps of Figure 17.

Step 1
This is handled at the time Bob creates an account through the UI.

Step 2
This is handled within the UI microservice (or within the api-gateway).

Step 3
This is handled within the api-gateway when a call is made by Alice to transfer to Bob.

Steps $4-5$
These steps are handled within `nf-token-controller.js`.

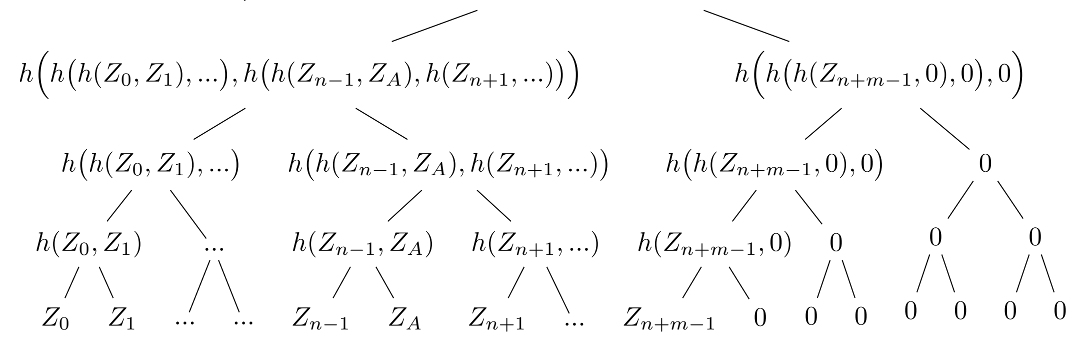
Steps $6-7$
These calls to the Shield contract are handled within `nf-token-zkp.js`.

It is important at this stage to note that there are an unknown number of other parties utilising the Shield contract. Hence, the dynamic array of tokens $\boldsymbol{Z}$ might have grown since Alice appended her Z_A as the n^{th} leaf of M (during the Mint explanation).
Suppose there have been $m-1$ additional tokens added to M since Alice added Z_A. That is,

$$\boldsymbol{Z}_{n+m-1} = (Z_0, Z_1, ..., Z_{n-1}, Z_A, Z_{n+1}, ..., Z_{n+m-1})$$

We denote the corresponding Merkle Tree which holds tokens $\boldsymbol{Z}_{n+m-1}$ by M_{n+m-1}. We denote its root by root_{n+m-1}; an element of $\textbf{\textsf{roots}} = (\mathsf{root}_0, \mathsf{root}_1, ..., \mathsf{root}_{n+m-1})$.

$$\mathsf{root}_{n+m-1} := h\Big(h\Big(h(h(Z_0, Z_1), ...), h(h(Z_{n-1}, Z_A), h(Z_{n+1}, ...))\Big), h\Big(h(h(Z_{n+m-1}, 0), 0), 0\Big)\Big)$$


$h\Big(h(h(Z_0, Z_1), ...), h(h(Z_{n-1}, Z_A), h(Z_{n+1}, ...))\Big)$
$h\Big(h(h(Z_{n+m-1}, 0), 0), 0\Big)$
$h\big(h(Z_0, Z_1), ...\big)$
$h\big(h(Z_{n-1}, Z_A), h(Z_{n+1}, ...)\big)$
$h\big(h(Z_{n+m-1}, 0), 0\big)$
0
$h(Z_0, Z_1)$
$...$
$h(Z_{n-1}, Z_A)$
$h(Z_{n+1}, ...)$
$h(Z_{n+m-1}, 0)$
0
0
0
Z_0
Z_1
$...$
$...$
Z_{n-1}
Z_A
Z_{n+1}
$...$
Z_{n+m-1}
0
0
0
0
0
0
0

Alice retrieves the value of the current Merkle root, root_{n+m-1}, from the Shield contract.

Since Alice knows that Z_A is at leaf-index n of M_{n+m-1}, Alice can also retrieve the path from the leaf $Z_n = Z_A$ to the root root_{n+m-1}. Path computations are done in `zkp/src/compute-vectors.js`.

We denote this path:

$$\phi_{Z_A} = [\phi_{d-1}, \phi_{d-2}, ..., \phi_1, \phi_0]$$

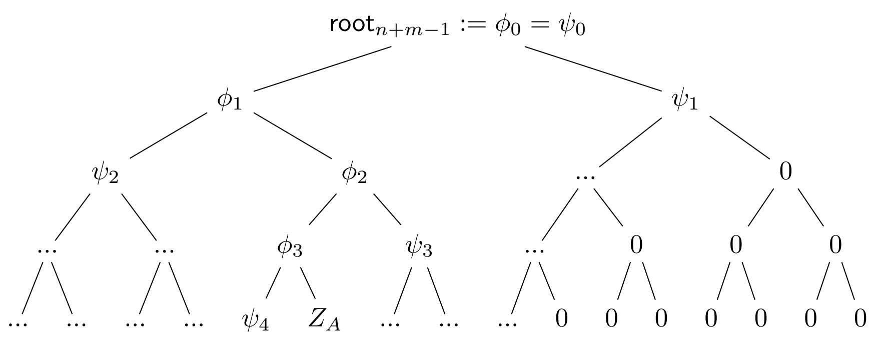
Note that $\phi_0 = \mathsf{root}_{n+m-1}$.

Alice also retrieve's the 'sister-path' of this path:

$$\psi_{Z_A} = [\psi_{d-1}, \psi_{d-2}, ..., \psi_1, \psi_0]$$

where $\psi_0 = \phi_0 = \mathsf{root}_{n+m-1}$

For ease of reading, let's focus only on the nodes of M_{n+m-1} which Alice cares about for the purposes of transferring to Bob:



$\mathsf{root}_{n+m-1} := \phi_0 = \psi_0$
ϕ_1
ψ_1
ψ_2
ϕ_2
...
0
...
...
ϕ_3
ψ_3
...
0
0
0
... ... ψ_4 Z_A 0 0 0 0 0 0 0

Equipped with ψ_{Z_A}, Alice can prove that she owns a token commitment at one of the leaves of M_{n+m-1}, without revealing that it is "Z_n located at leaf-index n".

Steps $8-9$
These steps are handled within `nf-token-controller.js`.

As a reminder, we let:

$x = (N_A,\ \mathsf{root}_{n+m-1},\ Z_B)$	Public Inputs used to generate the Proof
$\omega = (\alpha,\ \psi_{Z_A},\ sk_A,\ \sigma_A,\ pk_B,\ \sigma_{\vec{AB}})$	Private Inputs used to generate the Proof

Steps $10-11$
These steps are handled within a ZoKrates container.

Alice uses the $C_{nft-transfer}$ (or C) – the set of constraints for a non-fungible transfer, located in `zkp/code/gm17/nft-transfer` (see Trusted Setup). $C_{nft-transfer}(\ \omega,\ x\ )$ returns a value of *true* if Alice provides a set of valid 'satisfying' arguments (ω, x) to C.

Let's elaborate on each of the checks and calculations constraining the inputs to C (we highlight public inputs in **bold** below):

1. Calculate $h(sk_A) =: pk'_A$.
   Note that this newly calculated pk'_A should equal pk_A (Alice's public key), but we don't need to pass pk_A as a private input and explicitly check that $pk'_A = pk_A$; a check on the correctness of sk_A (and hence pk'_A) is implicitly achieved in the next two steps:

2. Calculate $h(\alpha \mid pk'_A \mid \sigma_A) =: Z'_A$.
   Note again that this newly calculated Z'_A should equal Z_A (Alice's token commitment), but we don't need to pass Z_A as a private input and explicitly check that $Z'_A = Z_A$; a check on the correctness of Z_A (and hence Z'_A) is implicitly achieved in the next step:

3. Check inputs $\psi_{Z_A} = [\psi_{d-1}, \psi_{d-2}, ..., \psi_1, \boldsymbol{\psi_0} = \mathbf{root_{n+m-1}}]$ and the newly calculated Z'_A satisfy:
   $h\Big(\psi_1 \,|...|\, h\Big(\psi_{d-2} \mid h\big(\psi_{d-1} \mid Z'_A\big)\Big)...\Big) = \mathsf{root}_{n+m-1} (=: \boldsymbol{\psi_0})$
   Given the one-way nature of our hashing function h, the only feasible way we could have arrived at the correct value of root_{n+m-1} is if the sister-path ψ_{Z_A} is correct, and if Z'_A is correct, which (working backwards) must mean that sk_A is correct.

   How does the circuit know the value of root_{n+m-1} is correct? It doesn't; but it is a 'public input', and we can rely upon the Shield smart contract to check the correctness of all public inputs.

   We've therefore shown in the steps so far, that:

   – Alice is the owner of a token commitment (because she knows its secret key)
   – Said token commitment is indeed a leaf of the on-chain Merkle Tree M_{n+m-1}.

   Alice commits to spending her token Z_A in the next step:

4. Check inputs $\sigma_A, sk_A, \boldsymbol{N_A}$ satisfy: $h(\sigma_A \mid sk_A) = \boldsymbol{N_A}$
   N_A is referred to as a 'nullifier' because it is understood by all participants to be an indisputable commitment to spend ('nullify') a token commitment. Remember that the token commitment being spent isn't revealed; the earlier steps have allowed Alice to demonstrate hidden knowledge of the secret key sk_A of a token commitment which does indeed exist. By including sk_A in the nullifier's preimage, Alice is binding herself as the executor of this transfer. By including σ_A, Alice is specifying a serial number which is unique to the token Z_A (thereby distinguishing this nullifier from those which would nullify any other token commitments she may own).

5. Check inputs $\alpha, pk_B, \sigma_{\vec{AB}}, \boldsymbol{Z_B}$ satisfy: $h(\alpha \mid pk_B \mid \sigma_{\vec{AB}}) = \boldsymbol{Z_B}$
   This final step constrains the same asset α to be included in Z_B as was included in Z_A.
   You might notice that the circuit doesn't actually constrain Alice to use the correct values for Bob's public key pk_B, nor the serial number $\sigma_{\vec{AB}}$ as inputs to the circuit. Alice is free to transfer ownership of the token commitment to anyone.

Notice how each stage is linked to the last, and that at each of the 'Check' stages, private inputs are being reconciled against at least one public input (highlighted in **bold** to help you notice). By structuring the circuit C in this way, we are able to share only the public inputs with the Shield contract (along with a 'proof' $\pi_{C,x,\omega}$). We'll see shortly that the Shield contract checks the correctness of each of the public inputs against its current states.

If all of the above constraints are satisfied by the public and private inputs, ZoKrates will generate the proof $\pi_{C,x,\omega}$; a proof of knowledge of satisfying arguments (ω, x) *s.t.* $C(\omega, x) = 1$.

Step 12
This transaction is handled within `nf-token-zkp.js`.

Having generated $\pi_{C,x,\omega}$, Alice then sends the following to the Shield contract from her anonymous Ethereum address $\Xi_{A,1}$:

$$\begin{aligned} &\pi_{C,x,\omega} \\ &x = (N_A, \mathsf{root}_{n+m-1}, Z_B) \end{aligned}$$

Recall that everyone knows the checks and calculations which have been performed in the circuit $C_{nft-transfer}$, because it is a public file in the Nightfall repository. Further, everyone knows the verification key vk_C which uniquely represents this circuit, because it has been publicly stored in the Verifier Registry contract. Therefore, when this anonymous caller (Alice) shares the pair $(x, \pi_{C,x,\omega})$, and the 'unique id' of the relevant verification key vk_C; everyone will interpret this information as the caller's intention to transfer, and everyone will be convinced that the caller knows the secret key which permits them to transfer ownership of a token commitment.

Steps 13 – 15
The Verifier Registry contract already has stored within it the verification key vk_C. It runs a verification function $V(vk_C, \pi_{C,x,\omega}, x)$.

$$V : (vk_C, \pi_{C,x,\omega}, x) \to \{0, 1\}$$

where:

$$V = \begin{cases} 1, & \text{if } \pi_{C,x,\omega} \text{ and } x \text{ satisfy } vk_C \\ 0, & \text{otherwise} \end{cases}$$

Steps 16 – 17
If the Verifier contract returns 1 (*true*) (verified) to the Shield contract, then the Shield contract will be satisfied that Alice's proof and public inputs represent her commitment to relinquish ownership of a token commitment, and to transfer ownership of the underlying asset to someone via the newly proposed token commitment Z_B. If the Verifier contract returns 0, then the transaction will revert.

Let's suppose Alice's $(x, \pi_{C,x,\omega})$ pair is verified.

Following verification of the proof, the Shield contract will do the following:

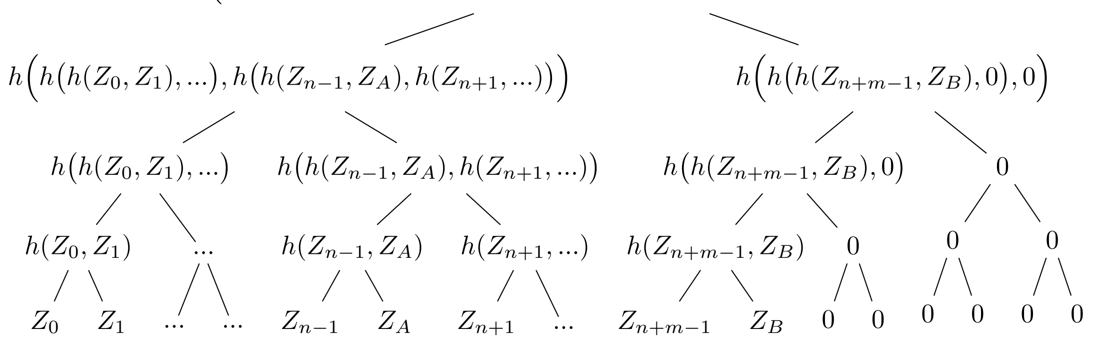
1. Check root_{n+m-1} is in **roots**.
   (If not, the transfer will fail)

2. Check N_A is not already in the list of nullifiers, which we denote $\boldsymbol{N}$.
   (If N_A is already in $\boldsymbol{N}$, the transfer will fail)

3. Append the commitment Z_B to the ever-increasing array of tokens, $\boldsymbol{Z}_{n+m}$, so that $\boldsymbol{Z}_{n+m} = (Z_0, Z_1, ...Z_{n-1}, Z_A, Z_{n+1}, ...Z_{n+}$

4. Recalculate a Merkle Root root_{n+m} of M_{n+m}

$$\mathsf{root}_{n+m} := h\Big(h\big(h(h(Z_0,Z_1),...),h(h(Z_{n-1},Z_A),h(Z_{n+1},...))\big),h\big(h(h(Z_{n+m-1},Z_B),0),0\big)\Big)$$



$h\big(h(h(Z_0,Z_1),...),h(h(Z_{n-1},Z_A),h(Z_{n+1},...))\big)$
$h\big(h(h(Z_{n+m-1},Z_B),0),0\big)$
$h\big(h(Z_0,Z_1),...\big)$
$h\big(h(Z_{n-1},Z_A),h(Z_{n+1},...)\big)$
$h\big(h(Z_{n+m-1},Z_B),0\big)$
0
$h(Z_0,Z_1)$
...
$h(Z_{n-1},Z_A)$
$h(Z_{n+1},...)$
$h(Z_{n+m-1},Z_B)$
0
0
0
Z_0 Z_1 ... ... Z_{n-1} Z_A Z_{n+1} ... Z_{n+m-1} Z_B 0 0 0 0 0 0

Note that the Shield contract only needs to calculate the hashes on the path from Z_B to the root.

5. Append root_{n+m} to the ever-increasing array **roots**

6. Similarly append the nullifier N_A to the ever-increasing array $\boldsymbol{N}$.

Steps 18 – 19
The api-gateway routes the data resulting from a transfer to her local database.

Similarly, the api-gateway ensures any sensitive data (data which is private to Alice alone) is filtered before Alice sends data to Bob.

Data which is crucial to Bob verifying his ownership of the new Z_B is encrypted with Bob's public whisper key pk_B^W and broadcast to the Whisper network.

Steps 20 – 21
Nightfall uses web3.shh to use Whisper. Bob's logged-in application will listen for all Whisper messages, and will try to decrypt all messages with his private whisper key sk_B^W. If decryption is successful, the data will be stored in the relevant database on Bob's local machine.

`nft-token-zkp.js` includes functions to cross-reference the data Bob has received from Alice against the data stored in the Shield contract.

Bob will store all important information in his private database.

7.3 Burn

We continue with the notation and indices from the prior sections.

Suppose Bob is the owner of the token commitment Z_B which represents the ERC-721 asset with tokenId α (as discussed in the prior section). The asset α can continue to be transferred under zero-knowledge between parties within the Shield contract indefinitely. Any third-party observers would not be able to infer "who sent what to whom".

Recall that whilst the ERC-721 token represented by α has a 'private' token commitment representation within the Shield contract, the underlying 'public' ERC-721 token is owned by the Shield contract; effectively 'locked up' in escrow.

Suppose Bob (now the owner of α because he knows the secret key $sk_{B,0}^{Z,(n+m+1)}$) wishes to 'release' his public ERC-721 token represented by α from escrow. Then he will need to effectively 'reveal' the contents of his token commitment Z_B in order to convince the Shield contract that he is indeed entitled to withdraw α from escrow. We call this act of converting from a 'private' token commitment back to its 'public' counterpart a '**burn**'.

Note that by burning a token commitment, Bob is revealing information which was previously private; namely, the asset α. Bob could continue to use an anonymous Ethereum address when calling the 'burn' transaction, but analytics of public ERC-721 transactions thereafter will likely eventually reveal that it was Bob who burned α. We'll have Bob use his public Ethereum address 'burn', for simplicity.

For Bob to burn Z_B within the Shield contract, under zero knowledge, he follows the steps in Figure 18.

Non-fungible burn algorithm

Bob's steps:

1. Compute $N_B := h(\ \sigma_{\vec{AB}} \mid sk_B^Z\ )$, the nullifier of Bob's commitment Z_B.
2. Get ψ_{Z_B} – the sister-path of Z_B – from the Shield contract (see Details below).
3. Get the latest Merkle root from the Shield contract: $\mathsf{root}_{n+m+k-1}$ (see Details below).
4. Set public inputs $x = (\alpha,\ N_B,\ \mathsf{root}_{n+m+k-1})$
5. Set private inputs $\omega = (\psi_{Z_A},\ sk_B,\ \sigma_{\vec{AB}})$
6. Select $C_{nft-burn}(\ \omega,\ x\ )$ – the set of constraints which are satisfied if and only if:
   (a) pk_B equals $h(\ sk_B\ )$; (Proof of knowledge of the secret key to pk_B) (see Details for why pk_B isn't an input to C)
   (b) Z_B equals $h(\ \alpha \mid pk_B \mid \sigma_{\vec{AB}}\ )$ (Proof of the constituent values of Z_B) (see Details for why Z_B isn't an input to C)
   (c) $\mathsf{root}_{n+m+k-1}$ equals $h\Big(\psi_1 \mid ... \mid h\Big(\psi_{d-2} \mid h\big(\psi_{d-1} \mid Z_B\big)\Big)...\Big)$ (Proof that Z_B belongs to the on-chain Merkle Tree)
   (d) N_B equals $h(\ \sigma_{\vec{AB}} \mid sk_B^Z\ )$ (Proof N_B is indeed the nullifier of Z_B)
7. Generate $\pi := P(\ p_C\ ,\ x,\ \omega\ )$; a proof of knowledge of satisfying arguments (ω, x) $s.t.$ $C(\omega, x) = 1$. Recall: p_C – the proving key for C – will be stored on Alice's computer.

   The pair (π, x) is the zk-SNARK which attests to knowledge of private inputs ω without revealing them.
8. Send (π, x) to the Shield contract for verification.

   Using web3: `nfTokenShield.burn(payTo, proof, inputs, vkId)`

   where `payTo` is an Ethereum address, specified by Bob, into which he wishes for the ERC-721 token with tokenId $= \alpha$ to be transferred.

Shield contract's steps:

9. Verify the proof as correct: call a Verifier contract to verify the `(proof, inputs)` pair against the verification key represented by `vkId`.

...

Figure 18a: Non-Fungible Burn Algorithm

Verifier contract's steps:

10. Compute `result = verify(proof, inputs, vkId)`.
    I.e. Verify the `(proof, inputs)` pair against the verification key.
11. Return `result`$\in\{$`false, true`$\}$ to the Shield contract.

Shield contract's steps:

12. If `result = false`, revert.
13. Else:
    (a) Check $\mathsf{root}_{n+m+k-1}$ is in **roots**. (Revert if not).
    (b) Check N_B is not already in its list of 'spent' nullifiers. (Revert if not).
    (c) Transfer the ERC-721 token with tokenId $= \alpha$ from the Shield contract (which has been holding it in escrow) to Bob's `payTo` Ethereum address.
    (d) Append the nullifier N_B to the ever-increasing array $\boldsymbol{N}$.

Bob's steps:

14. Check the ERC-721 contract to ensure he owns the token with tokenId $= \alpha$.
15. Store any relevant data in his local database.

Figure 18b: Non-Fungible Burn Algorithm

7.3.1 Details

We refer to the numbered steps of Figure 18.

Step 1
This is handled within `nf-token-controller.js`.

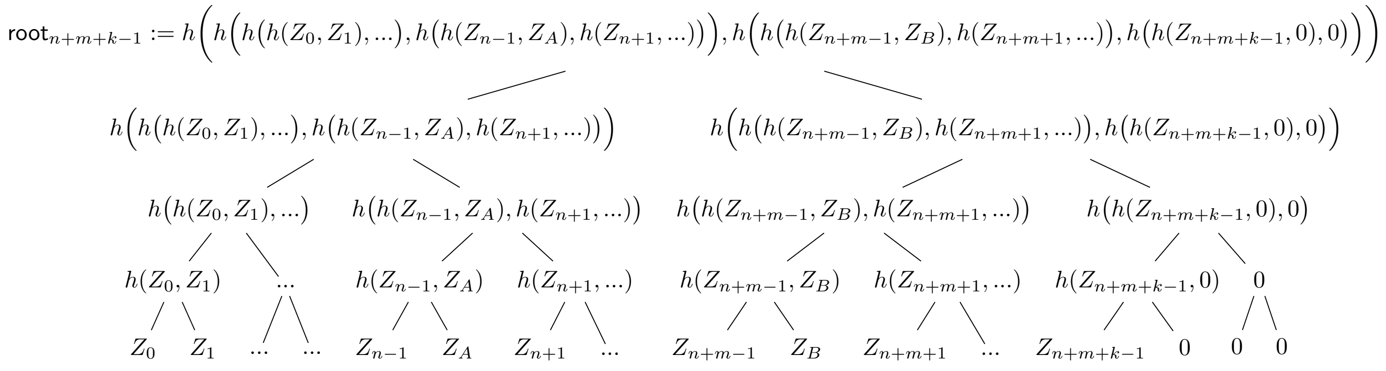
Steps $2-3$
These calls to the Shield contract are handled within `nf-token-zkp.js`.

It is important at this stage to note that there are an unknown number of other parties utilising the Shield smart contract. Hence, the dynamic array of tokens $\boldsymbol{Z}$ might have grown since Alice appended Bob's Z_B as the $(n+m)^{th}$ leaf of M.

Suppose there have been $k-1$ additional tokens added to $\boldsymbol{Z}$ since Alice added Bob's Z_B. That is,

$$\boldsymbol{Z}_{n+m+k-1} = (Z_0, Z_1, ..., Z_{n-1}, Z_A, Z_{n+1}, ..., Z_{n+m-1}, Z_B, Z_{n+m+1}, ..., Z_{n+m+k-1})$$

We denote the corresponding Merkle Tree which holds tokens $\boldsymbol{Z}_{n+m+k-1}$ by $M_{n+m+k-1}$. We denote its root by $\mathsf{root}_{n+m+k-1}$; an element of **roots**.



$\mathsf{root}_{n+m+k-1} := h\Big(h\big(h(h(Z_0,Z_1),...),h(h(Z_{n-1},Z_A),h(Z_{n+1},...))\big),h\big(h(h(Z_{n+m-1},Z_B),h(Z_{n+m+1},...)),h(h(Z_{n+m+k-1},0),0)\big)\Big)$
$h\big(h(h(Z_0,Z_1),...),h(h(Z_{n-1},Z_A),h(Z_{n+1},...))\big)$
$h\big(h(h(Z_{n+m-1},Z_B),h(Z_{n+m+1},...)),h(h(Z_{n+m+k-1},0),0)\big)$
$h(h(Z_0,Z_1),...)$
$h(h(Z_{n-1},Z_A),h(Z_{n+1},...))$
$h(h(Z_{n+m-1},Z_B),h(Z_{n+m+1},...))$
$h(h(Z_{n+m+k-1},0),0)$
$h(Z_0,Z_1)$
...
$h(Z_{n-1},Z_A)$
$h(Z_{n+1},...)$
$h(Z_{n+m-1},Z_B)$
$h(Z_{n+m+1},...)$
$h(Z_{n+m+k-1},0)$
0
Z_0
Z_1
...
...
Z_{n-1}
Z_A
Z_{n+1}
...
Z_{n+m-1}
Z_B
Z_{n+m+1}
...
$Z_{n+m+k-1}$
0
0
0

Bob retrieves the value of the current Merkle root, $\mathsf{root}_{n+m+k-1}$, from the Shield contract.

Since Bob knows that Z_B is at leaf-index $n+m$ of $M_{n+m+k-1}$, Bob can also retrieve the path from the leaf $Z_{n+m} = Z_B$ to the root $\mathsf{root}_{n+m+k-1}$. Path computations are done in `zkp/src/compute-vectors.js`.

We denote this path

$$\phi_{Z_B} = [\phi_{d-1}, \phi_{d-2}, ..., \phi_1, \phi_0]$$

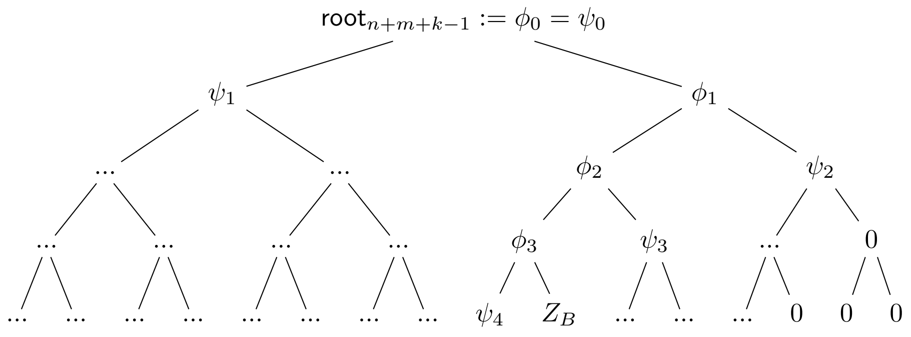
Note that $\phi_0 = \mathsf{root}_{n+m+k-1}$.

Bob also retrieve's the 'sister-path' of this path:

$$\psi_{Z_B} = [\psi_{d-1}, \psi_{d-2}, ..., \psi_1, \psi_0]$$

where $\psi_0 = \phi_0 = \mathsf{root}_{n+m+k-1}$.

For ease of reading, let's focus only on the nodes of $M_{n+m+k-1}$ which Bob cares about for the purposes of burning his token commitment Z_B:



Equipped with ψ_{Z_B}, Bob can prove that he owns a token commitment at one of the leaves of $M_{n+m+k-1}$, without revealing that it is "Z_{n+m} located at leaf-index $n+m$".

Steps $4-5$
These steps are handled within `nf-token-controller.js`.

As a reminder, we let:

$$\begin{array}{ll} x = (\alpha,\ N_B,\ \mathsf{root}_{n+m+k-1}) & \text{Public Inputs used to generate the Proof} \\ \omega = (\psi_{Z_B},\ sk_B,\ \sigma_{\vec{AB}}) & \text{Private Inputs used to generate the Proof} \end{array}$$

Steps $6-7$
These steps are handled within a ZoKrates container.

Bob uses the $C_{nft-burn}$ (or C) – the set of constraints for a non-fungible burn, located in `zkp/code/gm17/nft-burn` (see Trusted Setup). $C_{nft-burn}(\ \omega,\ x\ )$ returns a value of *true* if Bob provides a set of valid 'satisfying' arguments (ω, x) to C.

Let's elaborate on each of the checks and calculations constraining the inputs to C (we highlight public inputs in **bold** below):

1. Calculate $h(sk_B) =: pk'_B$.
   Note that this newly calculated pk'_B should equal pk_B (Bob's public key), but we don't need to pass pk_B as a private input and explicitly check that $pk'_B = pk_B$; a check on the correctness of sk_B (and hence pk'_B) is implicitly achieved in the next two steps:

2. Calculate $h(\boldsymbol{\alpha} \mid pk'_B \mid \sigma_{\vec{AB}}) =: Z'_B$.
   Note again that this newly calculated Z'_B should equal Z_B (Bob's token commitment), but we don't need to pass Z_B as a private input and explicitly check that $Z'_B = Z_B$; a check on the correctness of Z_B (and hence Z'_B) is implicitly achieved in the next step:

3. Check inputs $\psi_{Z_B} = [\psi_{d-1}, \psi_{d-2}, ..., \psi_1, \boldsymbol{\psi_0 = \mathsf{root}_{n+m+k-1}}]$ and the newly calculated Z'_B satisfy:
   $h\Big(\psi_1 \mid ... \mid h\Big(\psi_{d-2} \mid h(\psi_{d-1} \mid Z'_B)\Big)...\Big) = \mathsf{root}_{n+m+k-1} (=: \boldsymbol{\psi_0})$
   Given the one-way nature of our hashing function h, the only feasible way we could have arrived at the correct value of $\mathsf{root}_{n+m+k-1}$ is if the sister-path ψ_{Z_B} is correct, and if Z'_B is correct, which (working backwards) must mean that sk_B is correct.

   How does the circuit know the value of $\mathsf{root}_{n+m+k-1}$ is correct? It doesn't; but it is a 'public input', and we can rely upon the Shield smart contract to check the correctness of all public inputs.

   We've therefore shown in the steps so far, that:

   – Bob is the owner of a token commitment (because he knows its secret key)
   – Said token commitment is indeed a leaf of the on-chain Merkle Tree $M_{n+m+k-1}$.
   – The token commitment does indeed represent the ERC-721 token with tokenId $= \boldsymbol{\alpha}$ (remember that α is a public input to a 'burn' zk-SNARK).

   Bob commits to burning his token Z_B in the next step:

4. Check inputs $\sigma_{\vec{AB}}, sk_B, \boldsymbol{N_B}$ satisfy: $h(\sigma_{\vec{AB}} \mid sk_B) = \boldsymbol{N_B}$
   N_B is referred to as a 'nullifier' because it is understood by all participants to be an indisputable commitment to spend ('nullify') a token commitment. Remember that the token commitment being spent isn't revealed; the earlier steps have allowed Bob to demonstrate hidden knowledge of the secret key sk_B of a token commitment which does indeed exist. By including sk_B in the nullifier's preimage, Bob is binding himself as the executor of this 'burn'. By including $\sigma_{\vec{AB}}$, Bob is specifying a serial number which is unique to the token Z_B (thereby distinguishing this nullifier from those which would nullify any other token commitments he may own).

Notice how each stage is linked to the last, and that at each of the 'Check' stages, private inputs are being reconciled against at least one public input (highlighted in **bold** to help you notice). By structuring the circuit C in this way, we are able to share only the public inputs with the Shield contract (along with a 'proof' $\pi_{C,x,\omega}$). We'll see shortly that the Shield contract checks the correctness of each of the public inputs against its current states.

If all of the above constraints are satisfied by the public and private inputs, ZoKrates will generate the proof $\pi_{C,x,\omega}$; a proof of knowledge of satisfying arguments (ω, x) $s.t.$ $C(\omega, x) = 1$.

Step 8
This transaction is handled within `nf-token-zkp.js`.

Having generated $\pi_{C,x,\omega}$, Bob then sends the following to the Shield contract from his Ethereum address E_B:

$$E_B$$
$$\pi_{C,x,\omega}$$
$$x = (\alpha, N_B, \mathsf{root}_{n+m+k-1})$$

Recall that everyone knows the checks and calculations which have been performed in the circuit $C_{nft-burn}$, because it is a public file in the Nightfall repository. Further, everyone knows the verification key vk_C which uniquely represents this circuit, because it has been publicly stored in the Verifier Registry contract. Therefore, when Bob shares the pair $(x, \pi_{C,x,\omega})$, and the 'unique id' of the relevant verification key vk_C; everyone will interpret this information as the Bob's intention to burn; and everyone will be convinced that he knows the secret key which permits him to transfer ownership of a token commitment; and everyone will be convinced that that token commitment represents the ERC-721 token with tokenId $= \alpha$.

Steps 9 – 11
The Verifier Registry contract already has stored within it the verification key vk_C. It runs a verification function $V(vk_C, \pi_{C,x,\omega}, x)$.

$$V : (vk_C, \pi_{C,x,\omega}, x) \to \{0, 1\}$$

where:

$$V = \begin{cases} 1, & \text{if } \pi_{C,x,\omega} \text{ and } x \text{ satisfy } vk_C \\ 0, & \text{otherwise} \end{cases}$$

Steps 12 – 13
If the Verifier contract returns 1 (*true*) (verified) to the Shield contract, then the Shield contract will be satisfied that Bob's proof and public inputs represent his commitment to burning a token commitment, and to withdrawing its underlying ERC-721 token $= \alpha$. If the Verifier contract returns 0, then the transaction will revert.

Let's suppose Bob's $(x, \pi_{C,x,\omega})$ pair is verified.

Following verification of the proof, the Shield contract will do the following:

1. Check $\mathsf{root}_{n+m+k-1}$ is in **roots**.
   (If not, the burn will fail)
2. Check N_B is not already in the list of nullifiers, which we denote $\boldsymbol{N}$.
   (If N_B is already in $\boldsymbol{N}$, the burn will fail)
3. Transfer the ERC-721 token with tokenId $= \alpha$ from the Shield contract (i.e. from escrow) to Bob's Ethereum address.
4. Append the nullifier N_B to the ever-increasing array $\boldsymbol{N}$.

Steps 14 – 15
Bob is now the owner of the public ERC-721 token. The Nightfall UI queries the linked ERC-721 contract for tokens Bob owns. If Bob ever wished to convert this token back into a token commitment, he would need to do a non-fungible 'mint' (discussed earlier).

8 ERC-20 (fungible) Tokens

Contents

8.0.1 Variables 45
8.0.2 Key Management 46
8.1 Mint 47
8.1.1 Details 49
8.2 Transfer 51
8.2.1 Details 54
8.3 Burn 59
8.3.1 Details 62

We recommend reading the ERC-721 (non-fungible) Tokens protocol first, because non-fungibility makes things simpler.

In this section we'll give an overview of using Nightfall to privately transact fungible tokens (ft's). We'll cover three key functions:

- Mint - create an initial 'token commitment'; a private representation of a public ERC-20 token.
- Transfer - nullify the sender's token commitment, and generate a new token commitment to represent ownership by the recipient.
- Burn - nullify a token commitment, and receive the underlying public ERC-20 token.

8.0.1 Variables

Here we describe the variables used.

A, B	participants Alice and Bob.
pk_A	the public key belonging to Alice.
sk_A	the secret key belonging to Alice.
	Note: there are several (secret key, public key) pairs in this protocol)
E_A	the Ethereum address of Alice.
$\Xi_{A,i}$	an 'anonymous' Ethereum address belonging to Alice, where $i \in \mathbb{N}$ is an index, for distinguishing between multiple anonymous addresses.
$c, d, e, f \in \mathbb{R}^+\|_{dp}$	used to denote ERC-20 token values.
	Note that in practice, the accuracy of these values is restricted by the number of decimal places (dp) prescribed in the ERC-20 token contract.
σ	a 'salt' used to provide uniqueness to commitment preimages.
$\sigma_{\vec{AB}}$	stresses that a salt is being shared privately from Alice to Bob.
Z	An ERC-20 commitment; a zero-knowledge commitment representing ownership of an amount of ERC-20 tokens.
Z_A	Stresses that an ERC-20 commitment belongs to Alice.
Z_c	Stresses that an ERC-20 commitment represents a value of c (as denominated in the native currency of the ERC-20 token).
Z_l	Stresses that an ERC-20 commitment is the l^{th} leaf of a Merkle Tree (see below for M).
	Note that the meaning of these (seemingly colliding or ambiguous) subscripts will be clear from context.
N	A nullifier for an ERC-20 commitment Z.
N_A	A nullifier for the ERC-20 commitment Z_A
N_c	A nullifier for the ERC-20 commitment Z_c.
	Note that the meaning of these (seemingly colliding or ambiguous) subscripts will be clear from context.
M	A binary Merkle Tree.
M_l	A binary Merkle Tree with l non-zero leaves (where leaves are populated in order 'from left to right').
root_l	The root of M_l ('M' is omitted because context will be clear).
ϕ_L	$[\phi_L(d-1), \phi_L(d-2), ..., \phi_L(1), \phi_L(0)]$ - The path from a leaf L to the root of a Merkle Tree M, where $\phi_L(0) = \mathsf{root}$.
ϕ	$[\phi_{d-1}, \phi_{d-2}, ..., \phi_1, \phi_0]$ - Alternative notation for the path from a leaf, where the leaf L is clear from the context. $\phi_0 = \mathsf{root}$.
ψ_L	$[\psi_L(d-1), \psi_L(d-2), ..., \psi_L(1), \psi_L(0)]$ - The sister-path from a leaf L to the root of a Merkle Tree M, where $\psi_L(0) = \phi_L(0) = \mathsf{root}$.
ψ	$[\psi_{d-1}, \psi_{d-2}, ..., \psi_1, \psi_0]$ - Alternative notation for the sister-path from a leaf, where the leaf L is clear from the context. $\psi_0 = \mathsf{root}$.
x	Public inputs to a zk-SNARK.
ω	Private inputs to a zk-SNARK.
C	An arithmetic circuit $C : (\omega, x) \to \{0, 1\}$
p_C	A proving key for the circuit C.
vk_C	A verification key for the circuit C.
$\pi(p_C, x, \omega)$	A proof for the circuit C, public inputs x, and private inputs ω
$\pi_{C,x,\omega}$	An abbreviation of the above.
π	An abbreviation of the above, when the context of the proof is clear.
$h()$	A one-way hashing function. Nightfall currently uses sha256 hashing throughout.

8.0.2 Key Management

There are several pairs of public and private keys to keep track of throughout these protocols. We provide a summary here (for an actor Alice (A)):

Ownership of...	**Account symbol**	**Private Key**	**Public Key**	**Notes**
Ethereum address	E_A	sk_A^E	pk_A^E	Used for 'mint' and 'burn'.
Anonymous Ethereum addresses	$\Xi_{A,i}$	$sk_{A,i}^{\Xi}$	$pk_{A,i}^{\Xi}$	Used for 'transfer'. $i \in \mathbb{N}$.
Ethereum Whisper accounts	$W_{A,j}$	$sk_{A,j}^W$	$pk_{A,j}^W$	Used for private messaging. $j \in \mathbb{N}$
ERC-20 commitment Z_l	Z_l	$sk_A^{Z,l}$	$pk_A^{Z,l}$	Used to 'mint', 'transfer' and 'burn' Z_l.

Hereafter, when we write sk_A and pk_A we will be referring to $sk_A^{Z,l}$ and $pk_A^{Z,l}$ respectively (where Z_l is clear from context) - unless otherwise stated.

8.1 Mint

Suppose Alice wishes to be able to transfer ownership of an amount of ERC-20 tokens under zero-knowledge, so that the following become private:

1. The value of ERC-20 tokens being transferred.
2. The identity of the sender of the tokens ('Alice').
3. The identity of the recipient of the tokens.

In order to achieve this, Alice must first convert her ERC-20 tokens into a private ERC-20 commitment. We call this act of conversion **'minting'** an ERC-20 commitment.

In this section, we outline Nightfall's protocol for minting an ERC-20 commitment, but first, an important privacy warning is given in Figure 19.

PRIVACY WARNING

Privacy is NOT achieved during the minting stage!

Minting an ERC-20 commitment initially requires Alice to transfer a certain value of ERC-20 tokens to a 'Shield' contract (which thereafter holds this value in escrow). This transfer reveals the Ethereum address of the sender (Alice) as well as the value. Therefore everyone will know the owner and the underlying value being represented by the initial ERC-20 commitment which is created at this 'minting' stage.

Only during subsequent 'transfers' of the new ERC-20 commitment, will we achieve the privacy intentions of Figure **??**

Figure 19: Privacy warning: minting alone does not achieve privacy

Suppose Alice owns ERC-20 tokens of value c (denominated in the ERC-20 token's currency). Suppose Alice wishes to create a private token commitment, representing her ownership of value c.

For Alice to mint a token commitment Z_c representing value c on the blockchain, under zero knowledge, she follows the steps in Figure 20. Note: We avoid using Z_A (which was used to stress Alice's ownership in the non-fungible section), because Alice will own more than one token commitment when we explain 'transfers'.

Fungible mint algorithm

Alice's steps:

1. Generate a random salt σ_c.
2. Compute $Z_c := h(\ c \mid pk_A^Z \mid \sigma_c\ )$, a token commitment which represents c.
3. Set public inputs $x = (\ c,\ Z_c\ )$
4. Set private inputs $\omega = (\ pk_A^Z,\ \sigma_c\ )$
5. Select $C_{nft-mint}(\ \omega,\ x\ )$ – the set of constraints which are satisfied if and only if:
   (a) Z_c equals $h(\ c \mid pk_A^Z \mid \sigma_c\ )$ (Proof that the commitment Z_c hides the correct value c)
6. Generate $\pi := P(\ p_C\ ,\ x,\ \omega\ )$; a proof of knowledge of satisfying arguments (ω, x) $s.t.$ $C(\omega, x) = 1$.
   Recall: p_C – the proving key for C – will be stored on Alice's computer.

   The pair (π, x) is the zk-SNARK which attests to knowledge of private inputs ω without revealing them.
7. Send (π, x) to the Shield contract for verification.

   Using web3: `fTokenShield.mint(proof, inputs, vkId)`

Shield contract's steps:

8. Verify the proof as correct: call a Verifier contract to verify the `(proof, inputs)` pair against the verification key represented by `vkId`.

Verifier contract's steps:

9. Compute `result = verify(proof, inputs, vkId)`.

   I.e. Verify the `(proof, inputs)` pair against the verification key.
10. Return `result`$\in${`false, true`} to the Shield contract.

Shield contract's steps:

11. If `result = false`, revert.
12. Else:
    (a) Transfer a value of c, on behalf of Alice, to the Shield Contract. I.e. store c in escrow.
    (b) Add Z_c to the next empty leaf of the Merkle Tree.
    (c) Recalculate the path to the root of the Merkle Tree from Z_c for future users.

Alice's steps:

13. Store relevant data in local database, including the leafindex of Z_c.

Figure 20: Fungible Mint Algorithm

8.1.1 Details

We refer to the numbered steps of Figure 20.

Step 1
This is handled within the UI microservice (or within the api-gateway).

Steps 2 – 4
These steps are handled within `f-token-controller.js`.

Steps 5 – 6
These steps are handled within a ZoKrates container.

Step 7
This transaction is handled within `f-token-zkp.js`.

Steps 8 – 10
The Verifier contract already has stored within it the object vk_C (see Trusted Setup). It runs a verification function $V(vk_C, \pi, x)$.

$$V : (vk_C, \pi_{C,x,\omega}, x) \rightarrow \{0, 1\}$$

where:

$$V = \begin{cases} 1, & \text{if } \pi_{C,x,\omega} \text{ and } x \text{ satisfy } vk_C \\ 0, & \text{otherwise} \end{cases}$$

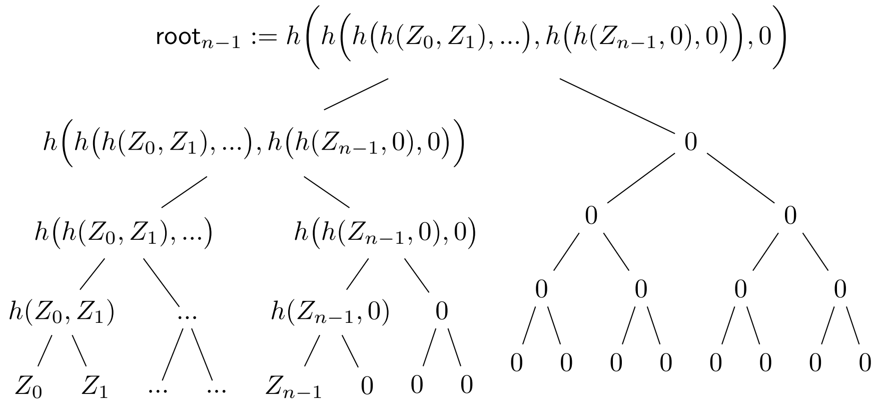
Steps 11 – 12
If the Verifier contract returns 1 (`true`) (verified) to the Shield contract, then the Shield contract will be satisfied with Alice's commitment, and will update its persistent states:

Suppose the Shield contract stores an ever-increasing array, $\boldsymbol{Z}$, of all token commitments which have ever been submitted by anyone.

Suppose, prior to Alice's mint, there are $n-1$ tokens in the tree:

$$\boldsymbol{Z}_{n-1} = (Z_0, Z_1, ..., Z_{n-1})$$

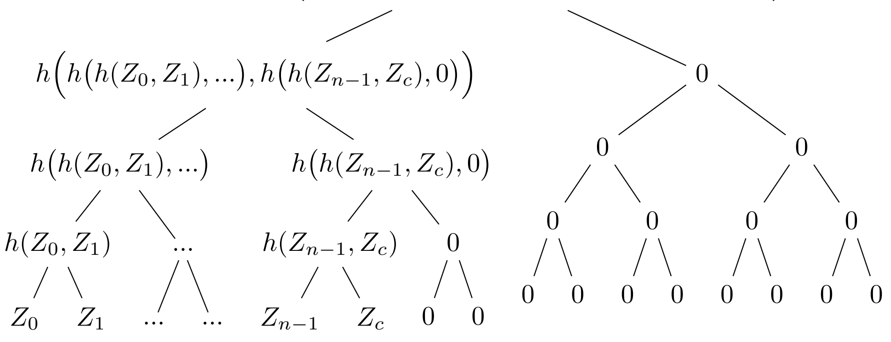
The information held within $\boldsymbol{Z}_{n-1}$ may be represented by the root hash root_{n-1} of a Merkle Tree M_{n-1}:


$\mathsf{root}_{n-1} := h\Big(h\big(h(h(Z_0, Z_1), ...), h(h(Z_{n-1}, 0), 0)\big), 0\Big)$
$h\big(h(h(Z_0, Z_1), ...), h(h(Z_{n-1}, 0), 0)\big)$
$h(h(Z_0, Z_1), ...)$
$h(h(Z_{n-1}, 0), 0)$
$h(Z_0, Z_1)$
...
$h(Z_{n-1}, 0)$
Z_0
Z_1
...
...
Z_{n-1}
0

Now that the Shield contract has been given verification that Alice's commitment, Z_c, does indeed hide the value c, the Shield contract will do the following:

- Append the commitment Z_c to the ever-increasing array of tokens, $\boldsymbol{Z}_{n-1}$, so that $\boldsymbol{Z}_n = (Z_0, Z_1, ...Z_{n-1}, Z_c)$
- Recalculate a Merkle Root root_n of M_n:

$$\mathsf{root}_n := h\Big(h\Big(h\big(h(Z_0, Z_1), ...\big), h\big(h(Z_{n-1}, Z_c), 0\big)\Big), 0\Big)$$



$h\Big(h\big(h(Z_0, Z_1), ...\big), h\big(h(Z_{n-1}, Z_c), 0\big)\Big)$
0
$h\big(h(Z_0, Z_1), ...\big)$
$h\big(h(Z_{n-1}, Z_c), 0\big)$
0
0
$h(Z_0, Z_1)$
...
$h(Z_{n-1}, Z_c)$
0
0 0 0 0
Z_0 Z_1 ... ...
Z_{n-1} Z_c 0 0
0 0 0 0 0 0 0 0

- Append root_n to an ever-increasing array $\textbf{roots} = (\mathsf{root}_0, \mathsf{root}_1, ..., \mathsf{root}_{n-1}, \mathsf{root}_n)$

Step 13
Alice will store all important information in her private database.

8.2 Transfer

We continue with the notation and indices from the 'Mint' section.

In the 'Mint' section, we saw how Alice can create an 'ERC-20 commitment' Z_c within the Shield contract which:

- hides an underlying value c, denominated in the currency of a particular ERC-20 contract; and
- hides and binds Alice as the owner of Z_c (and hence of value c) through an ownership keypair (sk_A^Z, pk_A^Z).

Recall our privacy intentions:
Alice wishes to be able to transfer ownership of an ERC-20 tokens under zero-knowledge, so that the following become private:

1. All details of the value being transacted.
2. The identity of the sender of the value ('Alice').
3. The identity of the recipient of the value.

Recall that minting a token commitment does not yet afford Alice any privacy (see the warning in Figure 19). Only with subsequent transfers will the whereabouts of value c and the amount Alice owns be hidden.

Suppose Alice wishes to transfer a value e to Bob under zero knowledge.

First, Alice must ensure she has 'minted' enough private token commitments which represent a value of at least e. For convenience, suppose Alice has minted two private token commitments, representing ownership of values c and d, where $c + d \geq e$. That is,

$$Z_c := h(\ c \mid pk_A^Z \mid \sigma_c\ )$$
$$Z_d := h(\ d \mid pk_A^Z \mid \sigma_d\ )$$

Let f be the balancing amount, so that $c + d = e + f$. In this example, we can think of f as Alice's 'change' when she pays Bob e.

Note that a fungible commitment transfer in Nightfall always requires **two** 'input' commitments and **two** 'output' commitments. There are several reasons for this:

- We're using zk-SNARKs to attest to proof of a 'transfer' computation. Due to the way a computation (a set of constraints) is abstracted into a (proving key, verification key) pair, the computations cannot be dynamically sized. That is, the number of variables (public and private inputs) being passed into the computation must be of a fixed size. Further, only fixed-sized for-loops are possible within the computation.

  Therefore, if we wanted to allow different permutations of 'number of inputs' and 'number of outputs', we would need to perform a trusted setup for each permutation; store the verification key for each on-chain; and distribute each proving key.

  To avoid such complexity at this stage, we have chosen "two inputs, two outputs" for now.
- Having just one output would mean the sender would have to own a set of commitments which sum to exactly the amount required by the recipient (no more, no less). This is impractical for most use cases.
- Having just one input increases the likelihood of an observer inferring information from analysis of transactions.

For Alice to transfer a value of e (and receive f as change) within the Shield contract, under zero knowledge, she follows the steps in Figure 21:

Fungible transfer algorithm

Bob's steps:

1. Before Alice can send him anything, Bob must register his public keys pk_B^Z and pk_B^W against both his public Ethereum address pk_B^E and his unique name 'Bob' within the PKD.

Alice's steps:

2. Generate new random salts σ_e and σ_f.
3. Lookup Bob's 'zkp' public key pk_B^Z from the PKD.
4. Compute $Z_e := h(\ e \mid pk_B^Z \mid \sigma_e\ )$, a token commitment which represents value e, to be owned by Bob.
5. Compute $Z_f := h(\ f \mid pk_A^Z \mid \sigma_f\ )$, a token commitment which represents value f, to be owned by Alice (as change).
6. Compute $N_c := h(\ \sigma_c \mid sk_A^Z\ )$, the nullifier of Alice's commitment Z_c.
7. Compute $N_d := h(\ \sigma_d \mid sk_A^Z\ )$, the nullifier of Alice's commitment Z_d.
8. Get ψ_{Z_c} – the sister-path of Z_c – from the Shield contract (see Details below).
9. Get ψ_{Z_d} – the sister-path of Z_d – from the Shield contract.
10. Get the latest Merkle root from the Shield contract: $\texttt{root}_{n+m+k-1}$ (see Details below).
11. Set public inputs $x = (\ N_c,\ N_d,\ Z_e,\ Z_f,\ \texttt{root}_{n+m+k-1})$
12. Set private inputs $\omega = (\ c,\ d,\ e,\ f,\ \psi_{Z_c},\ \psi_{Z_d},\ sk_A,\ \sigma_c,\ \sigma_d,\ pk_B,\ \sigma_e,\ \sigma_f)$
13. Select $C_{ft-transfer}(\ \omega,\ x\ )$ – the set of constraints which are satisfied if and only if:
    (a) pk_A equals $h(\ sk_A\ )$; (Proof of knowledge of the secret key to pk_A) (see Details for why pk_A isn't an input to C)
    (b) Z_c equals $h(\ c \mid pk_A \mid \sigma_c\ )$ (Proof of the constituent values of Z_c)
    (c) Z_d equals $h(\ d \mid pk_A \mid \sigma_d\ )$ (Proof of the constituent values of Z_c)
    (See Details for why Z_c and Z_d aren't inputs to C)
    (d) $\texttt{root}_{n+m+k-1}$ equals $h\Big(\psi_{Z_c}(1) \mid ... \mid h\Big(\psi_{Z_c}(d-2) \mid h\big(\psi_{Z_c}(d-1) \mid Z_c\big)\Big)...\Big)$ (Proof that Z_c belongs to the on-chain Merkle Tree)
    (e) $\texttt{root}_{n+m+k-1}$ equals $h\Big(\psi_{Z_d}(1) \mid ... \mid h\Big(\psi_{Z_d}(d-2) \mid h\big(\psi_{Z_d}(d-1) \mid Z_d\big)\Big)...\Big)$ (Proof that Z_d belongs to the on-chain Merkle Tree)
    (f) N_c equals $h(\ \sigma_c \mid sk_A^Z\ )$ (Proof that N_c is indeed the nullifier of Z_c)
    (g) N_d equals $h(\ \sigma_d \mid sk_A^Z\ )$ (Proof that N_d is indeed the nullifier of Z_d)
    (h) Z_e equals $h(\ e \mid pk_B^Z \mid \sigma_e\ )$ (Proof that Z_e hides value e)
    (i) Z_f equals $h(\ f \mid pk_A^Z \mid \sigma_e\ )$ (Proof that Z_f hides value f)
    (j) $c + d$ equals $e + f$.
    (k) The two most significant bits of each of c, d, e, f are both zero. This prevents the output values e and f from exceeding the maximum bit-lengths accepted by C (and hence prevents us from creating two unspendable commitments).
14. Generate $\pi := P(\ p_C\ ,\ x,\ \omega\ )$; a proof of knowledge of satisfying arguments (ω, x) *s.t.* $C(\omega, x) = 1$. Recall: p_C – the proving key for C – will be stored on Alice's computer.
    The pair (π, x) is the zk-SNARK which attests to knowledge of private inputs ω without revealing them.
15. Send (π, x) to the Shield contract for verification.
    Using web3: `nfTokenShield.transfer(proof, inputs, vkId)`

...

Figure 21a: Fungible Transfer Algorithm

Shield contract's steps:

16. Verify the proof as correct: call a Verifier contract to verify the `(proof, inputs)` pair against the verification key represented by `vkId`.

Verifier contract's steps:

17. Compute `result = verify(proof, inputs, vkId)`.
    I.e. Verify the `(proof, inputs)` pair against the verification key.
18. Return `result`$\in${`false, true`} to the Shield contract.

Shield contract's steps:

19. If `result = false`, revert.
20. Else:
    (a) Check $\mathsf{root}_{n+m+k-1}$ is in **roots**. (Revert if not).
    (b) Check N_c is not already in its list of 'spent' nullifiers. (Revert if not).
    (c) Check N_d is not already in its list of 'spent' nullifiers. (Revert if not).
    (d) Add Z_e to the next empty leaf of the Merkle Tree.
    (e) Recalculate the path to the root of the Merkle Tree from Z_e for future users.
    (f) Add Z_f to the next empty leaf of the Merkle Tree.
    (g) Recalculate the path to the root of the Merkle Tree from Z_f for future users.
    (h) Append the newly calculated root root_{n+m+k} to the ever-increasing array **roots**
    (i) Similarly append the nullifiers N_c and N_d to the ever-increasing array $\boldsymbol{N}$.

Alice's steps:

21. Store relevant data in her local database, including the leaf-indices of Z_e and Z_f.
22. Send Bob important data privately via Whisper (using his public key pk_B^W):
    (a) The salt σ_e of Z_e.
    (b) The public key of Bob, pk_B^Z, used by Alice in the preimage of Z_e (for completeness, so Bob can check the correctness of Z_e himself).
    (c) The value e.
    (d) Z_e.
    (e) The leaf-index of Z_e within the on-chain Merkle Tree M (so Bob can locate it).

Bob's steps:

23. Check the correctness of the information provided by Alice:
    (a) Check Z_e equals $h(\ e \mid pk_B^Z \mid \sigma_e\ )$
    (b) Check that Z_e is stored at the leafIndex of M which Alice claimed.
24. Store relevant data in his local database, including whether or not his 'correctness checks' passed.

Figure 21b: Fungible Transfer Algorithm

8.2.1 Details

We refer to the numbered steps of Figure 17.

Step 1
This is handled at the time Bob creates an account through the UI.

Step 2
This is handled within the UI microservice (or within the api-gateway).

Step 3
This is handled within the api-gateway when a call is made by Alice to transfer to Bob.

Steps $4-7$
These steps are handled within `nf-token-controller.js`.

Steps $8-10$
These calls to the Shield contract are handled within `nf-token-zkp.js`.

It is important at this stage to note that there are an unknown number of other parties utilising the Shield contract. Hence, the dynamic array of tokens $\boldsymbol{Z}$ might have grown since Alice appended her Z_c as the n^{th} leaf of M (during the Mint explanation).

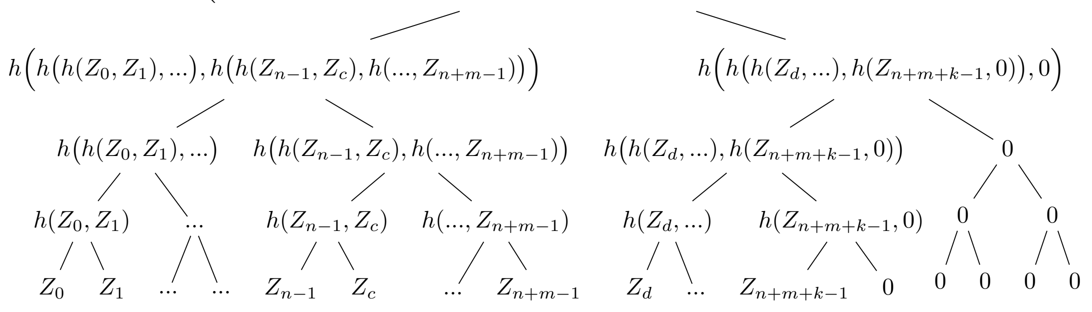
Suppose Z_c is located at the leaf-index n of the merkle tree M (and hence can also be denoted Z_n) and Z_d is located at the leaf-index $n+m$ of the merkle tree M.

Suppose there have been $k-1$ additional tokens added to M since Alice added Z_d. That is,

$$\boldsymbol{Z}_{n+m+k-1} = (Z_0, Z_1, ..., Z_{n-1}, Z_c, ..., Z_{n+m-1}, Z_d, ..., Z_{n+m+k-1})$$

We denote the corresponding Merkle Tree which holds tokens $\boldsymbol{Z}_{n+m+k-1}$ by $M_{n+m+k-1}$. We denote its root by $\mathsf{root}_{n+m+k-1}$; an element of $\textbf{roots} = (\mathsf{root}_0, \mathsf{root}_1, ..., \mathsf{root}_{n+m+k-1})$.

$$\mathsf{root}_{n+m+k-1} := h\Big(h\Big(h(h(Z_0,Z_1),...),h(h(Z_{n-1},Z_c),h(...,Z_{n+m-1}))\Big),h\Big(h(h(Z_d,...),h(Z_{n+m+k-1},0)),0\Big)\Big)$$



$h\Big(h(h(Z_0,Z_1),...),h(h(Z_{n-1},Z_c),h(...,Z_{n+m-1}))\Big)$
$h\Big(h(h(Z_d,...),h(Z_{n+m+k-1},0)),0\Big)$
$h(h(Z_0,Z_1),...)$
$h(h(Z_{n-1},Z_c),h(...,Z_{n+m-1}))$
$h(h(Z_d,...),h(Z_{n+m+k-1},0))$
0
$h(Z_0,Z_1)$
...
$h(Z_{n-1},Z_c)$
$h(...,Z_{n+m-1})$
$h(Z_d,...)$
$h(Z_{n+m+k-1},0)$
0
0
Z_0
Z_1
...
...
Z_{n-1}
Z_c
...
Z_{n+m-1}
Z_d
...
$Z_{n+m+k-1}$
0
0
0
0
0

Alice retrieves the value of the current Merkle root, $\mathsf{root}_{n+m+k-1}$, from the Shield contract.

Since she knows the index of Z_c is n within the leaves of $M_{n+m+k-1}$, Alice can also retrieve from $M_{n+m+k-1}$ the nodes of the path from the leaf Z_n to the root $\mathsf{root}_{n+m+k-1}$. We denote this path:

$$\phi_{Z_c} = [\phi_{Z_c}(d-1), \phi_{Z_c}(d-2), ..., \phi_{Z_c}(1), \phi_{Z_c}(0)]$$

Note that $\phi_{Z_c}(0) = \mathsf{root}_{n+m+k-1}$.

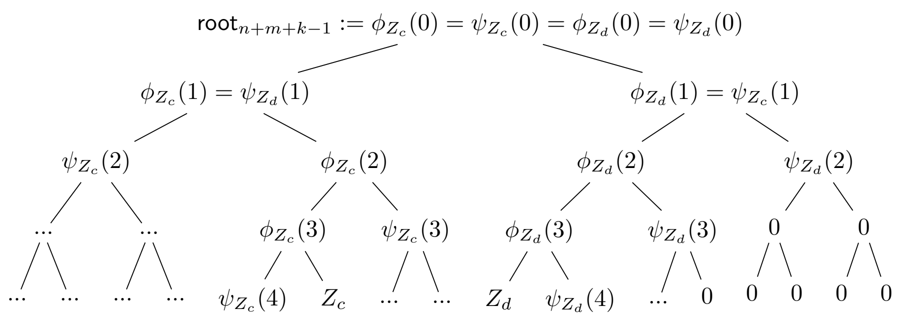
Alice also retrieve's the 'sister-path' of this path:

$$\psi_{Z_c} = [\psi_{Z_c}(d-1), \psi_{Z_c}(d-2), ..., \psi_{Z_c}(1), \psi_{Z_c}(0)]$$

where $\psi_{Z_c}(0) = \phi_{Z_c}(0) = \mathsf{root}_{n+m+k-1}$.

Similarly, Alice retrieves the path and sister-path ϕ_{Z_d} and ψ_{Z_d}.

For ease of reading, let's focus only on the nodes of M_{n+m-1} which Alice cares about for the purposes of transferring to Bob:



Equipped with ψ_{Z_c} and ψ_{Z_d}, Alice can prove knowledge of leaves Z_c and Z_d in $M_{n+m+k-1}$ without revealing Z_c, Z_d, nor the paths ϕ_{Z_c}, ψ_{Z_c}, ϕ_{Z_d}, ψ_{Z_d}; all she needs to reveal is the root $\mathsf{root}_{n+m+k-1}$ along with her proof.

Steps $11 - 12$
These steps are handled within `f-token-controller.js`.

As a reminder, we let:

$x = (N_c,\ N_d,\ Z_e,\ Z_f,\ \mathsf{root}_{n+m+k-1})$	Public Inputs used to generate the Proof
$\boldsymbol{\omega} = (c, d, e, f, \psi_{Z_c},\ \psi_{Z_d},\ sk_A,\ \sigma_c,\ \sigma_d,\ pk_B,\ \sigma_e,\ \sigma_f)$	Private Inputs used to generate the Proof

Steps $13 - 14$
These steps are handled within a ZoKrates container.

Alice uses the $C_{ft-transfer}$ (or C) – the set of constraints for a fungible transfer, located in `zkp/code/gm17/ft-transfer` (see Trusted Setup). $C_{ft-transfer}(\ \omega,\ x\ )$ returns a value of *true* if Alice provides a set of valid 'satisfying' arguments (ω, x) to C.

Let's elaborate on each of the checks and calculations constraining the inputs to C (we highlight public inputs in **bold** below):

1. Calculate $h(sk_A) =: pk'_A$.
   Note that this newly calculated pk'_A should equal pk_A (Alice's public key), but we don't need to pass pk_A as a private input and explicitly check that $pk'_A = pk_A$; a check on the correctness of sk_A (and hence pk'_A) is implicitly achieved in the next four steps:
2. Calculate $h(c \mid pk'_A \mid \sigma_c) =: Z'_c$.
3. Calculate $h(d \mid pk'_A \mid \sigma_d) =: Z'_d$.
   Note again that these newly calculated Z'_c and Z'_d values should equal Z_c and Z_d respectively (Alice's token commitments). But we don't need to pass Z_c and Z_d as private inputs and explicitly check that $Z'_c = Z_c$ and $Z'_d = Z_d$; a check on the correctness of Z'_c and Z'_d is implicitly achieved in the next step:
4. Check inputs $\psi_{Z_c} = [\psi_{Z_c}(d-1), \psi_{Z_c}(d-2), ..., \psi_{Z_c}(1), \boldsymbol{\psi_{Z_c}(0) = \mathsf{root}_{n+m+k-1}}]$ and the newly calculated Z'_c satisfy:
   $h\Big(\psi_{Z_c}(1) \mid ... \mid h\Big(\psi_{Z_c}(d-2) \mid h\big(\psi_{Z_c}(d-1) \mid Z'_c\big)\ \Big)...\Big) = \mathsf{root}_{n+m+k-1} (=: \boldsymbol{\psi_{Z_c}(0)})$
   Given the one-way nature of our hashing function h, the only feasible way we could have arrived at the correct value of $\mathsf{root}_{n+m+k-1}$ is if the sister-path ψ_{Z_c} is correct, and if Z'_c is correct, which (working backwards) must mean that sk_A is correct.

5. Check inputs $\psi_{Z_d} = [\psi_{Z_d}(d-1), \psi_{Z_d}(d-2), ..., \psi_{Z_d}(1), \boldsymbol{\psi_{Z_d}(0)} = \mathbf{root_{n+m+k-1}}]$ and the newly calculated Z'_d satisfy:
   $h\Big(\psi_{Z_d}(1) \mid ... \mid h\Big(\psi_{Z_d}(d-2) \mid h\big(\psi_{Z_d}(d-1) \mid Z'_d\big)\Big)...\Big) = \mathsf{root}_{n+m+k-1} (=: \boldsymbol{\psi_{Z_d}(0)})$
   Given the one-way nature of our hashing function h, the only feasible way we could have arrived at the correct value of $\mathsf{root}_{n+m+k-1}$ is if the sister-path ψ_{Z_d} is correct, and if Z'_d is correct, which (working backwards) must mean that sk_A is correct.

   How does the circuit know the value of $\mathsf{root}_{n+m+k-1}$ is correct? It doesn't; but it is a 'public input', and we can rely upon the Shield smart contract to check the correctness of all public inputs.

   We've therefore shown in the steps so far, that:

   – Alice is the owner of two token commitments (because she knows their secret key)
   – Said token commitments are indeed leaves of the on-chain Merkle Tree $M_{n+m+k-1}$.

   Alice commits to spending her tokens Z_c and Z_d in the next two steps:

6. Check inputs $\sigma_c, sk_A, \boldsymbol{N_c}$ satisfy: $h(\sigma_c \mid sk_A) = \boldsymbol{N_c}$
   N_c is referred to as a 'nullifier' because it is understood by all participants to be an indisputable commitment to spend ('nullify') a token commitment. Remember that the token commitment being spent isn't revealed; the earlier steps have allowed Alice to demonstrate hidden knowledge of the secret key sk_A of a token commitment which does indeed exist. By including sk_A in the nullifier's preimage, Alice is binding herself as the executor of this transfer. By including σ_c, Alice is specifying a serial number which is unique to the token Z_c (thereby distinguishing this nullifier from those which would nullify any other token commitments she may own).

7. Check inputs $\sigma_d, sk_A, \boldsymbol{N_d}$ satisfy: $h(\sigma_d \mid sk_A) = \boldsymbol{N_d}$

8. Check inputs $e, pk_B, \sigma_e, \boldsymbol{Z_e}$ satisfy: $h(e \mid pk_B \mid \sigma_e) = \boldsymbol{Z_B}$
   This step constrains a value of e to be included in Z_e.

9. Check inputs $f, \sigma_f, \boldsymbol{Z_f}$ and the calculated pk'_A satisfy: $h(f \mid pk'_A \mid \sigma_f) = \boldsymbol{Z_f}$
   This step constrains a value of f to be included in Z_f. Note that the default constraints for a fungible transfer in Nightfall force the second output token commitment to be returned to the sender as 'change', as we're forcing pk'_A to be a constituent of Z_f. It would be straightforward to make edits to allow this second token commitment to be transferred to anyone.

10. Check that $c + d = e + f$. We must constrain that no value is created or lost.

11. Check that the most significant bit of each of c, d, e, f is **zero**. This prevents either (or both) of e and f from exceeding the maximum bit-length of input values to C. If these bit-lengths were exceeded, then when if we were to attempt to transfer Z_e and Z_f in future, they would be rejected by C.

Notice how each stage is linked to the last, and that at each of the 'Check' stages, private inputs are being reconciled against at least one public input (highlighted in **bold** to help you notice). By structuring the circuit C in this way, we are able to share only the public inputs with the Shield contract (along with a 'proof' $\pi_{C,x,\omega}$). We'll see shortly that the Shield contract checks the correctness of each of the public inputs against its current states.

If all of the above constraints are satisfied by the public and private inputs, ZoKrates will generate the proof $\pi_{C,x,\omega}$; a proof of knowledge of satisfying arguments (ω, x) $s.t.$ $C(\omega, x) = 1$.

Step 15
This transaction is handled within `f-token-zkp.js`.

Having generated $\pi_{C,x,\omega}$, Alice then sends the following to the Shield contract from her anonymous Ethereum address $\Xi_{A,1}$:

$$\pi_{C,x,\omega}$$
$$x = (\ N_c,\ N_d,\ Z_e,\ Z_f,\ \mathsf{root}_{n+m+k-1})$$

Recall that everyone knows the checks and calculations which have been performed in the circuit $C_{ft-transfer}$, because it is a public file in the Nightfall repository. Further, everyone knows the verification key vk_C which uniquely represents this circuit, because it has been publicly stored in the Verifier Registry contract. Therefore, when this anonymous caller (Alice) shares the pair $(x, \pi_{C,x,\omega})$, and the 'unique id' of the relevant verification key vk_C; everyone will interpret this information as the caller's intention to transfer, and everyone will be convinced that the caller knows the secret key which permits them to transfer ownership of a token commitment.

Steps $16 - 18$
The Verifier Registry contract already has stored within it the verification key vk_C. It runs a verification function $V(vk_C, \pi_{C,x,\omega}, x)$.

$$V : (vk_C, \pi_{C,x,\omega}, x) \rightarrow \{0, 1\}$$

where:

$$V = \begin{cases} 1, & \text{if } \pi_{C,x,\omega} \text{ and } x \text{ satisfy } vk_C \\ 0, & \text{otherwise} \end{cases}$$

Steps $19 - 20$
If the Verifier contract returns 1 (*true*) (verified) to the Shield contract, then the Shield contract will be satisfied that Alice's proof and public inputs represent her commitment to relinquish ownership of two token commitments (which only Alice knows to be Z_c and Z_d), and to transfer ownership of value to someone via the newly proposed token commitment Z_e (whilst receiving 'change' in the form of Z_f). If the Verifier contract returns 0, then the transaction will revert.

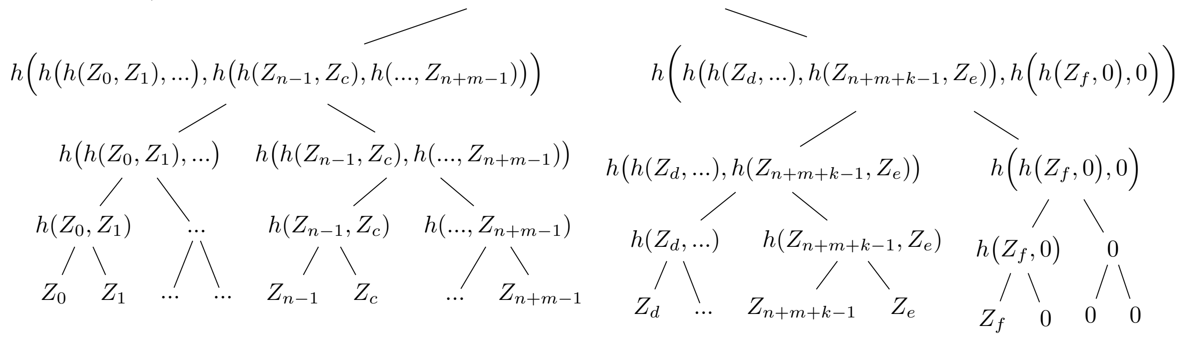
Let's suppose Alice's $(x, \pi_{C,x,\omega})$ pair is verified.

Following verification of the proof, the Shield contract will do the following:

1. Check $\mathsf{root}_{n+m+k-1}$ is in **roots**.
   (If not, the transfer will fail)
2. Check N_c is not already in the list of nullifiers, which we denote $\boldsymbol{N}$.
   (If N_c is already in $\boldsymbol{N}$, the transfer will fail)
3. Check N_d is not already in the list of nullifiers, which we denote $\boldsymbol{N}$.
   (If N_d is already in $\boldsymbol{N}$, the transfer will fail)
4. Append the commitment Z_e to the ever-increasing array of tokens, $\boldsymbol{Z}_{n+m+k}$, so that $\boldsymbol{Z}_{n+m+k} = (Z_0, Z_1, ...Z_{n-1}, Z_c, ..., Z_d, ...$
5. Recalculate a Merkle Root root_{n+m+k} of M_{n+m+k}
6. Append the commitment Z_f to the ever-increasing array of tokens, $\boldsymbol{Z}_{n+m+k+1}$, so that $\boldsymbol{Z}_{n+m+k} = (Z_0, Z_1, ...Z_{n-1}, Z_c, ..., Z_d,$
7. Recalculate a Merkle Root $\mathsf{root}_{n+m+k+1}$ of $M_{n+m+k+1}$

   Note: The 'recalculation' of the Merkle Root within the Shield contract only recalculates the hashes on the path from the newly added leaf to the root. We do not recalculate the entire Merkle Tree as that would require exponentially more computations. Hence, we need to perform this 'recalculation' each time a new commitment is added as a leaf.

$$\mathsf{root}_{n+m+k+1} := h\Bigg(h\Big(h\big(h(Z_0, Z_1), ...\big), h\big(h(Z_{n-1}, Z_c), h(..., Z_{n+m-1})\big)\Big), h\Big(h\big(h(Z_d, ...), h(Z_{n+m+k-1}, Z_e)\big), h\big(h(Z_f, 0), 0\big)\Big)\Bigg)$$


$h\Big(h\big(h(Z_0, Z_1), ...\big), h\big(h(Z_{n-1}, Z_c), h(..., Z_{n+m-1})\big)\Big)$
$h\Big(h\big(h(Z_d, ...), h(Z_{n+m+k-1}, Z_e)\big), h\big(h(Z_f, 0), 0\big)\Big)$
$h\big(h(Z_0, Z_1), ...\big)$
$h\big(h(Z_{n-1}, Z_c), h(..., Z_{n+m-1})\big)$
$h\big(h(Z_d, ...), h(Z_{n+m+k-1}, Z_e)\big)$
$h\big(h(Z_f, 0), 0\big)$
$h(Z_0, Z_1)$
...
$h(Z_{n-1}, Z_c)$
$h(..., Z_{n+m-1})$
$h(Z_d, ...)$
$h(Z_{n+m+k-1}, Z_e)$
$h(Z_f, 0)$
0
Z_0
Z_1
...
...
Z_{n-1}
Z_c
...
Z_{n+m-1}
Z_d
...
$Z_{n+m+k-1}$
Z_e
Z_f
0
0
0

8. Append $\mathsf{root}_{n+m+k+1}$ to the ever-increasing array **roots**
9. Similarly append the nullifiers N_c and N_d to the ever-increasing array $\boldsymbol{N}$.

Steps $21 - 22$
Nightfall uses web3.shh to use Whisper. Bob's logged-in application will listen for all Whisper messages, and will try to decrypt all messages with his private whisper key sk_B^W. If decryption is successful, the data will be stored in the relevant database on Bob's local machine.

`ft-token-zkp.js` includes functions to cross-reference the data Bob has received from Alice against the data stored in the Shield contract.

Bob will store all important information in his private database.

8.3 Burn

We continue with the notation and indices from the prior sections.

Suppose Bob is the owner of the token commitment Z_e which represents a value of e (denominated in the currency of a particular ERC-20 token).

Recall that whilst the token commitment Z_e exists (read: "is spendable") within the Shield contract, the Shield contract holds an equivalent value of e 'public' ERC-20 tokens; effectively 'locked up' in escrow.

Suppose Bob (now the owner of Z_e because he knows the secret key sk_B) wishes to 'release' a value of e ERC-20 tokens from escrow. Then he will need to effectively 'reveal' the contents of his token commitment Z_e in order to convince the Shield contract that he is indeed entitled to withdraw e from escrow. We call this act of converting from a 'private' token commitment back to its 'public' counterpart a '**burn**'.
Note that by burning a token commitment, Bob is revealing information which was previously private; namely, the value e. Bob could continue to use an anonymous Ethereum address when calling the 'burn' transaction, but analytics of public ERC-20 transactions thereafer will likely eventually reveal that it was Bob who burned e. We'll have Bob use his public Ethereum address 'burn', for simplicity.

For Bob to burn Z_e within the Shield contract, under zero knowledge, he follows the steps in Figure 22.

Fungible burn algorithm

Bob's steps:

1. Compute $N_e := h(\ \sigma_e \mid sk_B^Z\ )$, the nullifier of Bob's commitment Z_e.
2. Get ψ_{Z_e} – the sister-path of Z_e – from the Shield contract (see Details below).
3. Get the latest Merkle root from the Shield contract: $\mathsf{root}_{n+m+k+l}$ (see Details below).
4. Set public inputs $x = (\ e,\ N_e,\ \mathsf{root}_{n+m+k+l})$
5. Set private inputs $\omega = (\psi_{Z_e},\ sk_B,\ \sigma_e)$
6. Select $C_{ft-burn}(\ \omega,\ x\ )$ – the set of constraints which are satisfied if and only if:
   (a) pk_B equals $h(\ sk_B\ )$; (Proof of knowledge of the secret key to pk_B) (see Details for why pk_B isn't an input to C)
   (b) Z_e equals $h(\ e \mid pk_B \mid \sigma_e\ )$ (Proof of the constituent values of Z_c) (See Details for why Z_e isn't an input to C)
   (c) $\mathsf{root}_{n+m+k+l}$ equals $h\Big(\psi_{Z_e}(1) \mid ... \mid h\Big(\psi_{Z_e}(d-2) \mid h\big(\psi_{Z_e}(d-1) \mid Z_e\big)\Big) ...\Big)$ (Proof that Z_e belongs to the on-chain Merkle Tree)
   (d) N_e equals $h(\ \sigma_e \mid sk_B^Z\ )$ (Proof that N_e is indeed the nullifier of Z_e)
7. Generate $\pi := P(\ p_C\ ,\ x,\ \omega\ )$; a proof of knowledge of satisfying arguments (ω, x) $s.t.$ $C(\omega, x) = 1$. Recall: p_C – the proving key for C – will be stored on Alice's computer.

   The pair (π, x) is the zk-SNARK which attests to knowledge of private inputs ω without revealing them.
8. Send (π, x) to the Shield contract for verification.

   Using web3: `fTokenShield.burn(payTo, proof, inputs, vkId)`

   where `payTo` is an Ethereum address, specified by Bob, into which he wishes for e to be transferred (denominated in the currency of the linked ERC-20 contract).

Shield contract's steps:

9. Verify the proof as correct: call a Verifier contract to verify the `(proof, inputs)` pair against the verification key represented by `vkId`.

...

Figure 22a: Fungible Burn Algorithm

Verifier contract's steps:

10. Compute `result = verify(proof, inputs, vkId)`.
    I.e. Verify the `(proof, inputs)` pair against the verification key.
11. Return `result`$\in${`false, true`} to the Shield contract.

Shield contract's steps:

12. If `result = false`, revert.
13. Else:
    (a) Check $\mathsf{root}_{n+m+k+l}$ is in **roots**. (Revert if not).
    (b) Check N_e is not already in its list of 'spent' nullifiers. (Revert if not).
    (c) Transfer ERC-20 tokens of value e from the Shield contract (which has been holding the value in escrow) to Bob's `payTo` Ethereum address.
    (d) Append the nullifier N_e to the ever-increasing array $\boldsymbol{N}$.

Bob's steps:

14. Check the ERC-20 contract to ensure his balance has increased by e.
15. Store any relevant data in his local database.

Figure 22b: Fungible Burn Algorithm

8.3.1 Details

We refer to the numbered steps of Figure 22.

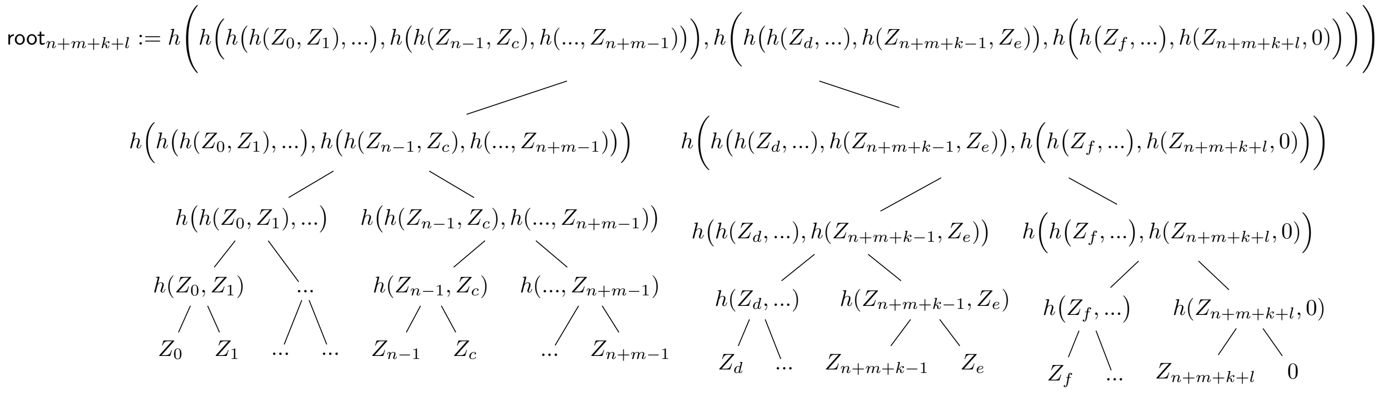
Step 1
This is handled within `f-token-controller.js`.

Steps $2-3$
These calls to the Shield contract are handled within `f-token-zkp.js`.

It is important at this stage to note that there are an unknown number of other parties utilising the Shield smart contract. Hence, the dynamic array of tokens $\boldsymbol{Z}$ might have grown since Alice appended Bob's Z_e and Alice's Z_f as the $(n+m)^{th}$ and $(n+m+1)^{th}$ leaves of M. Suppose there have been $l-1$ additional tokens added to $\boldsymbol{Z}$ since then. That is,

$$\boldsymbol{Z}_{n+m+k-1} = (Z_0, Z_1, ..., Z_{n-1}, Z_c, ..., Z_{n+m-1}, Z_d, ..., Z_{n+m+k-1}, Z_e, Z_f, Z_{n+m+k+2}, ...Z_{n+m+k+l})$$

We denote the corresponding Merkle Tree which holds tokens $\boldsymbol{Z}_{n+m+k+l}$ by $M_{n+m+k+l}$. We denote its root by $\mathsf{root}_{n+m+k+l}$; an element of **roots**.



$\mathsf{root}_{n+m+k+l} := h\Big(h\big(h(h(Z_0,Z_1),...),h(h(Z_{n-1},Z_c),h(...,Z_{n+m-1}))\big),h\Big(h(h(Z_d,...),h(Z_{n+m+k-1},Z_e)),h\big(h(Z_f,...),h(Z_{n+m+k+l},0)\big)\Big)\Big)$
$h\big(h(h(Z_0,Z_1),...),h(h(Z_{n-1},Z_c),h(...,Z_{n+m-1}))\big)$
$h\Big(h(h(Z_d,...),h(Z_{n+m+k-1},Z_e)),h\big(h(Z_f,...),h(Z_{n+m+k+l},0)\big)\Big)$
$h(h(Z_0,Z_1),...)$
$h(h(Z_{n-1},Z_c),h(...,Z_{n+m-1}))$
$h(h(Z_d,...),h(Z_{n+m+k-1},Z_e))$
$h\big(h(Z_f,...),h(Z_{n+m+k+l},0)\big)$
$h(Z_0,Z_1)$
...
$h(Z_{n-1},Z_c)$
$h(...,Z_{n+m-1})$
$h(Z_d,...)$
$h(Z_{n+m+k-1},Z_e)$
$h(Z_f,...)$
$h(Z_{n+m+k+l},0)$
Z_0
Z_1
...
...
Z_{n-1}
Z_c
...
Z_{n+m-1}
Z_d
...
$Z_{n+m+k-1}$
Z_e
Z_f
...
$Z_{n+m+k+l}$
0

Bob retrieves the value of the current Merkle root, $\mathsf{root}_{n+m+k+l}$, from the Shield contract.

Since Bob knows that Z_e is at leaf-index $n+m+k$ of $M_{n+m+k+l}$, Bob can also retrieve the path from the leaf $Z_{n+m+k} = Z_e$ to the root $\mathsf{root}_{n+m+k+l}$. Path computations are done in `zkp/src/compute-vectors.js`.

We denote this path

$$\phi_{Z_e} = [\phi_{d-1}, \phi_{d-2}, ..., \phi_1, \phi_0]$$

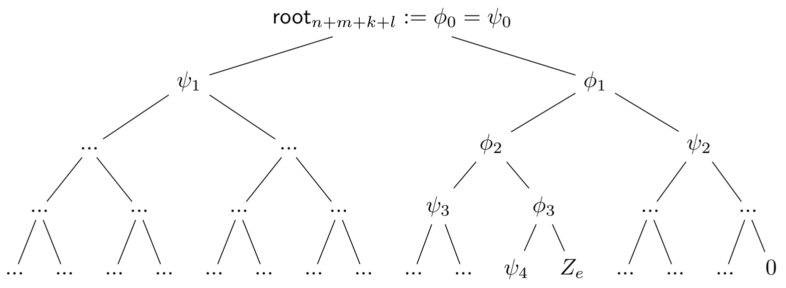
Note that $\phi_0 = \mathsf{root}_{n+m+k+l}$.

Bob also retrieve's the 'sister-path' of this path:

$$\psi_{Z_e} = [\psi_{d-1}, \psi_{d-2}, ..., \psi_1, \psi_0]$$

where $\psi_0 = \phi_0 = \mathsf{root}_{n+m+k+l}$.

For ease of reading, let's focus only on the nodes of $M_{n+m+k+1}$ which Bob cares about for the purposes of burning his token commitment Z_e:



Equipped with ψ_{Z_e}, Bob can prove that he owns a token commitment at one of the leaves of $M_{n+m+k+l}$, without revealing that it is "Z_{n+m+k} located at leaf-index $n+m+k$".

Steps $4-5$
These steps are handled within `f-token-controller.js`.

As a reminder, we let:

$$\begin{aligned} x &= (e,\ N_e,\ \mathsf{root}_{n+m+k+l}) && \text{Public Inputs used to generate the Proof} \\ \omega &= (\psi_{Z_e},\ sk_B,\ \sigma_e) && \text{Private Inputs used to generate the Proof} \end{aligned}$$

Steps $6-7$
These steps are handled within a ZoKrates container.

Bob uses the $C_{ft-burn}$ (or C) – the set of constraints for a fungible burn, located in `zkp/code/gm17/ft-burn` (see Trusted Setup). $C_{ft-burn}(\ \omega,\ x\ )$ returns a value of *true* if Bob provides a set of valid 'satisfying' arguments (ω, x) to C.

Let's elaborate on each of the checks and calculations constraining the inputs to C (we highlight public inputs in **bold** below):

1. Calculate $h(sk_B) =: pk'_B$.
   Note that this newly calculated pk'_B should equal pk_B (Bob's public key), but we don't need to pass pk_B as a private input and explicitly check that $pk'_B = pk_B$; a check on the correctness of sk_B (and hence pk'_B) is implicitly achieved in the next two steps:

2. Calculate $h(\boldsymbol{e} \mid pk'_B \mid \sigma_e) =: Z'_e$.
   Note again that this newly calculated Z'_e should equal Z_e (Bob's token commitment), but we don't need to pass Z_e as a private input and explicitly check that $Z'_e = Z_e$; a check on the correctness of Z_e (and hence Z'_e) is implicitly achieved in the next step:

3. Check inputs $\psi_{Z_e} = [\psi_{d-1}, \psi_{d-2}, ..., \psi_1, \boldsymbol{\psi_0 = \mathsf{root}_{n+m+k+l}}]$ and the newly calculated Z'_e satisfy:
   $h\Big(\psi_1 \mid ... \mid h\Big(\psi_{d-2} \mid h(\psi_{d-1} \mid Z'_B)\Big)...\Big) = \mathsf{root}_{n+m+k+l} (=: \boldsymbol{\psi_0})$
   Given the one-way nature of our hashing function h, the only feasible way we could have arrived at the correct value of $\mathsf{root}_{n+m+k+l}$ is if the sister-path ψ_{Z_e} is correct, and if Z'_e is correct, which (working backwards) must mean that sk_B is correct.

   How does the circuit know the value of $\mathsf{root}_{n+m+k+l}$ is correct? It doesn't; but it is a 'public input', and we can rely upon the Shield smart contract to check the correctness of all public inputs.

   We've therefore shown in the steps so far, that:

   – Bob is the owner of a token commitment (because he knows its secret key)
   – Said token commitment is indeed a leaf of the on-chain Merkle Tree $M_{n+m+k+l}$.
   – The token commitment does indeed represent a value of e ERC-20 tokens (remember that e is a public input to a 'burn' zk-SNARK).

   Bob commits to burning his token Z_e in the next step:

4. Check inputs $\sigma_e, sk_B, \boldsymbol{N_e}$ satisfy: $h(\sigma_e \mid sk_B) = \boldsymbol{N_e}$
   N_e is referred to as a 'nullifier' because it is understood by all participants to be an indisputable commitment to spend ('nullify') a token commitment. Remember that the token commitment being spent isn't revealed; the earlier steps have allowed Bob to demonstrate hidden knowledge of the secret key sk_B of a token commitment which does indeed exist. By including sk_B in the nullifier's preimage, Bob is binding himself as the executor of this 'burn'. By including σ_e, Bob is specifying a serial number which is unique to the token Z_e (thereby distinguishing this nullifier from those which would nullify any other token commitments he may own).

Notice how each stage is linked to the last, and that at each of the 'Check' stages, private inputs are being reconciled against at least one public input (highlighted in **bold** to help you notice). By structuring the circuit C in this way, we are able to share only the public inputs with the Shield contract (along with a 'proof' $\pi_{C,x,\omega}$). We'll see shortly that the Shield contract checks the correctness of each of the public inputs against its current states.

If all of the above constraints are satisfied by the public and private inputs, ZoKrates will generate the proof $\pi_{C,x,\omega}$; a proof of knowledge of satisfying arguments (ω, x) $s.t.$ $C(\omega, x) = 1$.

Step 8
This transaction is handled within `f-token-zkp.js`.

Having generated $\pi_{C,x,\omega}$, Bob then sends the following to the Shield contract from his Ethereum address E_B:

$$E_B$$
$$\pi_{C,x,\omega}$$
$$x = (e, N_e, \mathsf{root}_{n+m+k+l})$$

Recall that everyone knows the checks and calculations which have been performed in the circuit $C_{ft-burn}$, because it is a public file in the Nightfall repository. Further, everyone knows the verification key vk_C which uniquely represents this circuit, because it has been publicly stored in the Verifier Registry contract. Therefore, when Bob shares the pair $(x, \pi_{C,x,\omega})$, and the 'unique id' of the relevant verification key vk_C; everyone will interpret this information as the Bob's intention to burn; and everyone will be convinced that he knows the secret key which permits him to transfer ownership of a token commitment; and everyone will be convinced that that token commitment represents a value of e ERC-20 tokens.

Steps 9 – 11
The Verifier Registry contract already has stored within it the verification key vk_C. It runs a verification function $V(vk_C, \pi_{C,x,\omega}, x)$.

$$V : (vk_C, \pi_{C,x,\omega}, x) \to \{0, 1\}$$

where:

$$V = \begin{cases} 1, & \text{if } \pi_{C,x,\omega} \text{ and } x \text{ satisfy } vk_C \\ 0, & \text{otherwise} \end{cases}$$

Steps 12 – 13
If the Verifier contract returns 1 (*true*) (verified) to the Shield contract, then the Shield contract will be satisfied that Bob's proof and public inputs represent his commitment to burning a token commitment, and to withdrawing its underlying ERC-721 token $= \alpha$. If the Verifier contract returns 0, then the transaction will revert.

Let's suppose Bob's $(x, \pi_{C,x,\omega})$ pair is verified.

Following verification of the proof, the Shield contract will do the following:

1. Check $\mathsf{root}_{n+m+k+l}$ is in **roots**.
   (If not, the burn will fail)
2. Check N_e is not already in the list of nullifiers, which we denote $\boldsymbol{N}$.
   (If N_e is already in $\boldsymbol{N}$, the burn will fail)
3. Transfer a value of e ERC-20 tokens from the Shield contract (i.e. from escrow) to Bob's Ethereum address.
4. Append the nullifier N_e to the ever-increasing array $\boldsymbol{N}$.

Steps 14 – 15
Bob is now the owner of e more ERC-20 tokens. The Nightfall UI queries the linked ERC-20 contract for tokens Bob owns. If Bob ever wished to convert some or all of this value back into a token commitment, he would need to do a fungible 'mint' (discussed earlier).

References

[1] EY Global Blockchain R&D. Nightfall: Zokrates library for private token transfer over the ethereum blockchain. https://github.com/EYBlockchain/nightfall.

[2] Zcash. What are zk-SNARKs? https://z.cash/technology/zksnarks/.

[3] Vitalik Buterin. zk-SNARKS: Under the hood. https://medium.com/@VitalikButerin/zk-snarks-under-the-hood-b33151a013f6.

[4] Christian Reitwiessner. zk-SNARKs in a nutshell. https://blog.ethereum.org/2016/12/05/zksnarks-in-a-nutshell/.

[5] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part II*, pages 581–612, 2017.

[6] Stefan Deml Jacob Eberhardt, Thibaut Schaeffer. ZoKrates. https://github.com/Zokrates/ZoKrates.

[7] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: nearly practical verifiable computation. *Commun. ACM*, 59(2):103–112, 2016.