ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

web3
foundation

Confidential Cross-Blockchain Exchanges: Designing a Privacy-Preserving Interoperability Scheme

Master Thesis

A. Sánchez

December 28, 2020

Advisors: Prof. Dr. M. Vechev[1], Dr. P. Tsankov[1]
External advisors: Dr. F. Shirazi[2], Dr. A. Stewart[2]

[1]Department of Computer Science, ETH Zürich
[2]Web3 Foundation, Zug

Abstract

Interoperability solutions for privacy-focused cryptocurrencies are hard to design: they must integrate with the unique privacy features of the underlying blockchain, itself often in active development, such as to trace specific payments in protocols designed to make payments untraceable. The required specialised research and development effort have resulted in a lack of such projects, despite the dependence on centralised exchanges for cross-chain transfers being particularly problematic for privacy-oriented cryptocurrencies.

In this work, we present ZCLAIM, an adaptation of the framework for decentralised cross-chain exchanges XCLAIM to the privacy-protecting cryptocurrency Zcash. This is to the author's knowledge the first project that succeeds in maintaining privacy in cross-chain transfers.

ZCLAIM integrates with the Sapling version of the Zcash protocol, implemented on a smart-contract capable issuing chain, in order to attain private cross-chain transfers. We provide an abstract protocol specification, defining new transfer types that fit into the existing Sapling transaction structure along with zk-SNARKs allowing protocol participants to prove statements about Zcash transactions in zero knowledge on the issuing chain.

Acknowledgements

This thesis has been an incredible journey, taking more turns and allowing me to learn more than I could have ever expected.

Right by my side in this journey were Fatemeh Shirazi and Alistair Stewart, my two supervisors at the Web3 Foundation, to whom I am immensely grateful for their guidance, help and support throughout the thesis. I am also grateful to everyone else at the foundation for their interest and eagerness to help, and for all absolutely enjoyable off-topic chats. You truly made me feel at home.

I would also not be handing this in today if it were not for Petar Tsankov, who introduced me to the people at the Web3 Foundation and kindly offered to supervise me from the ETH side. I am grateful for his time, support and patience while he helped me figure it out, and for the vital directions and advice concerning the thesis structure with which he provided me.

Moreover, I would like to extend my gratitude to Professor Martin Vechev for giving me the opportunity to write my thesis at the SRI Lab.

Last but not least, I would like to thank my family, friends and flatmates for supporting me and brightening my mood during the most stressful times. If you are reading this: you are the best.

Contents

Chapter 1

Introduction

Since the creation of Bitcoin in 2009, thousands of cryptocurrencies and blockchains with a wide array of applications have emerged [1]. This fragmentation of the blockchain space has led to a high demand for interoperability, primarily in the form of asset transfers, across blockchains. Unfortunately, the siloed design of the vast majority of blockchains makes it difficult to design generic, cross-chain trading protocols, hence this challenge has to date been solved mostly through centralised exchanges.

However, these services require trust and undermine anonymity, which is particularly problematic when exchanging assets to or from privacy-oriented cryptocurrencies. Such projects, also termed privacy coins, are designed around the notion of payment anonymity and have built-in privacy features that obfuscate user activity. It is also these very features that make it especially challenging to design trustless cross-chain trading protocols involving privacy coins.

1.1 Interoperability

When talking about interoperability across blockchains nowadays, most of the time one type or another of cross-chain asset transfers is meant. Although there have been attempts at cross-chain smart contracts [2] and they have recently started being discussed in the literature [3], we still have a long way to go until that extent of interoperability becomes commonplace.

The classical solution to decentralised cross-chain exchanges are Atomic Cross-Chain Swaps (ACCS) [4]. Atomic swaps allow users on two different blockchains to swap ownership of a pre-agreed amout of assets, guaranteeing that the exchange either happens in full or not at all. Atomic swaps work, but they also present significant limitations such as requiring to establish an external communication channel between participants in order

to find and agree on a swap, an asymmetrical advantage for one out of the two participants ('free-option problem' [5]) and relatively high cost and long confirmation delays.

Most other protocols that have not been designed in this manner can be classified as *asset migration* protocols [6], in which an asset is moved from one blockchain to another. Usually, this is achieved by creating a representation of the assets on the new chain while those on the original chain are locked until the process is reversed. This representation of the locked assets on the alien chain is often referred to as a *tokenised representation* of the original assets, *wrapped tokens* or *cryptocurrency-backed assets (*CbAs*)*.

Besides, asset migration protocols may be classified as relying on a trusted third party (TTP) or as being *trustless*. Usually, relying on a TTP requires sacrificing control over one's funds and privacy, while trustless protocols require an economic system on the remote chain.

1.2 Related Work

Decentralised exchanges (DEXs) [7, 8] offer a trustless alternative to centralised cryptocurrency exchanges. However, most DEXs do not enable cross-chain transfers, but only serve to exchange assets within the blockchain on which they are deployed. The Ren Project [5] is a generic cross-chain transfer protocol aiming to implement universal interoperability, which they define as 'the ability to send any asset from any chain to any other chain for use in any application'. This is an impressive proposition, yet it is another question whether the issue of preserving privacy in cross-chain exchanges with privacy coins will be addressed in that context.

As far as cross-chain protocols involving privacy coins are concerned, a number of projects are currently under development. Cosmos [9] is working on a Zcash 'pegzone' [10], with the ultimate goal of enabling 'shielded transfers from the pegzone to Zcash and vice versa'. Details on the project are scarce, but as per this stated purpose, the Zcash pegzone aims to achieve very much the same goal as this work.

Wrapped [11] is a tokenised representation of Zcash on Ethereum, although it relies on a centralised TTP, does not support shielded Zcash and hence can by no means be considered to preserve privacy. Similarly, the Ren Project mentioned above offers a tokenised representation of ZCash, renZEC, which does not rely on a TTP but also does not support Zcash's shielded payment scheme.

Finally, there is an ongoing effort to implement atomic swaps between Bitcoin and the privacy coin Monero [12, 13].

1.3 Contributions

We introduce ZCLAIM, a protocol enabling the creation of fully private CBAs backed by the privacy coin Zcash. This work is originally based on XCLAIM, itself a framework laying out the creation of tokens on a pre-existing, smart-contract capable *issuing chain* backed by assets on a *backing chain*, both of them publicly auditable.

In contrast to these assumptions, we observe the case where:

(a) the backing chain is private, i.e. transactions leak no significant amount of information to outsiders,

(b) this anonymity shall be retained throughout the protocol and across both chains, and

(c) the definition of the issuing chain is loosened: it may be a smart-contract capable blockchain or a blockchain with appropriate functionality, built expressly to implement this protocol.

Chapter 2

Background

We present here an overview of mechanisms and terminology used in both XCLAIM and Zcash. This section is not meant to be an in-depth examination of these systems, but merely to provide a basic understanding of the components therein, most importantly of those central to ZCLAIM. For more material on the subjects, we refer the reader to the XCLAIM research paper [14], the original Zerocash paper [15, 16], of which Zcash is an implementation, and the Zcash Protocol Specification (for the Sapling upgrade in specific, for which ZCLAIM has been designed) [17].

2.1 XCLAIM

XCLAIM is a generic passive-mode interoperability framework for cross-chain exchanges using CBAs. It leverages smart contract logic, a dynamic set of economically incentivised, trustless intermediaries and cross-chain state verification to achieve decentralised asset interoperability.

2.1.1 Setup

In simple terms, XCLAIM allows users to create CBAs on an *issuing chain* I backed by funds on a *backing chain* B, and to redeem them again for backing currency at any moment. The *backing currency* on B is denoted by b, the *native currency* on I by i and the issued CBAs by $i(b)$. In a slight abuse of notation, b, $i(b)$ and i may also refer to quantities in these currencies where appropriate. $|x|$ denotes the monetary value of an amount of currency x.

The requirements on blockchains B and I are defined formally in [14, Section VI-B], but for the purpose of this review, we may assume that I may be any smart-contract capable blockchain while all blockchains qualify as B.

Following actors take an active role in the protocol:

- **Requesters** lock b on B to request the corresponding amount of $i(b)$ on I.
- **Redeemers** destroy $i(b)$ on I to request the corresponding amount of b on B. The redeemer of a certain amount of funds is not necessarily the same entity as their requester.
- **Vaults** are the non-trusted intermediaries that act as custodians, safekeep locked funds on B and are liable for fulfilling redeem requests of $i(b)$ for b on B. Anyone can take on the role of a vault by locking some collateral in i and registering as a vault. Vaults are incentivised by fees they derive from transactions in which they take part. In case of misbehaviour, they face *slashing*, which means partial or total liquidation of their collateral.
- The **issuing Smart Contract** (iSC) is a public smart contract responsible for managing the correct issuing of $i(b)$ on I and for ensuring the correct behaviour of vaults. It maintains a public list of vaults along with the amount of collateral they have locked, their identities on B and the amount of b that they hold.

  The iSC verifies transactions on B through a chain relay [18, 19], which stores and maintains block headers from blocks in B on I. It is assumed each block header contains the root of a Merkle tree containing all transactions or transaction identifiers for that block. The chain relay allows the iSC to verify that a transaction has been included on B if a Merkle path from the transaction to this root is provided. We call this an *inclusion proof*. Furthermore, the chain relay is able to verify if consensus has been reached on a certain block in B. How this is achieved depends on the consensus mechanism used in B.

2.1.2 Blockchain model and assumptions

XClaim assumes that the consensus mechanisms employed by B and I cannot be corrupted by an adversary, in order to ensure safety and liveness of the underlying blockchains. The specific assumptions depend on the consensus mechanism of the respective blockchain. For example, if B or I employ the Nakamoto consensus mechanism, such as Bitcoin [20] or Ethereum [21], it is assumed that the computational power of an adversary is limited by $\alpha < 33\%$. For Proof-of-Stake mechanisms with similar Byzantine fault tolerance, such as the upcoming Ethereum 2.0 upgrade [22] or [23, 24], it is assumed $f < n/3$, where n is the total number of consensus participants. Note that other Proof-of-Stake systems, such as [25], may assign voting power based on the staked amount or other criteria instead of per participant. In this case, it would be accurate to define f as the fraction of voting power held by the adversary.

This guarantees that the fraction of maliciously generated blocks are upper bounded by $\frac{f}{n-f}$ and hence that the probability of a blockchain reorganisation drops exponentially with security parameter $k \in \mathbb{N}$. k denotes the block depth, i.e., the position of a block relative to the tip of the blockchain. A transaction is considered '*securely* included in the underlying blockchain if, given the current blockchain head at position $h \in \mathbb{N}$, the transaction is included in a block at position $j \in \mathbb{N}$, such that $h - j \geq k$' [14, p. 3]. The security parameter k is denoted k^B for B and k^I for I.

Δ^I and Δ^B denote the delay from transaction broadcast to its secure inclusion on blockchains I and B, respectively. Δ_{submit} is the delay from block creation on B to the time a transaction submitting its block header to the chain relay is broadcast.

To manage exchange rate fluctuations between b and i, XCLAIM assumes an oracle $\mathcal{O}$ provides the iSC with the exchange rate $\varepsilon_{(i,b)} \in \mathbb{R}_{\geq 0}$ such that $1\, i = \varepsilon_{(i,b)}\, b$.

For the underlying network, it is assumed '(i) honest nodes are well connected and (ii) communication channels between these nodes are (semi-)-synchronous. Specifically, transactions broadcast by users are received by (honest) consensus participants within a known maximum delay Δ_{tx}' [14, p. 3].

2.1.3 System goals

XCLAIM achieves a number of security properties, defined as follows in [14, Section III-E]:

- **Auditability.** Any user with read access to blockchain B and I can audit the operation of XCLAIM and detect protocol failures.
- **Consistency.** No CBA units $i(b)$ can be issued without the equivalent amount of backing currency b being locked, i.e., that $|b| = |i(b)|$.
- **Redeemability.** Any user can redeem CBAs $i(b)$ for backing currency b on B, or be reimbursed with equivalent economic value on I.
- **Liveness.** Any user in XCLAIM can issue, transfer and swap CBAs without requiring a third party, i.e., liveness relies only on the secure operation of B and I.
- **Atomic Swaps.** Users can atomically swap XCLAIM CBAs against other assets on I or the native currency i.

2.1.4 Protocols

There are two main protocols in XCLAIM: *Issue*, allowing the creation of $i(b)$, and *Redeem*, which involves *burning* (destroying) $i(b)$ in exchange for

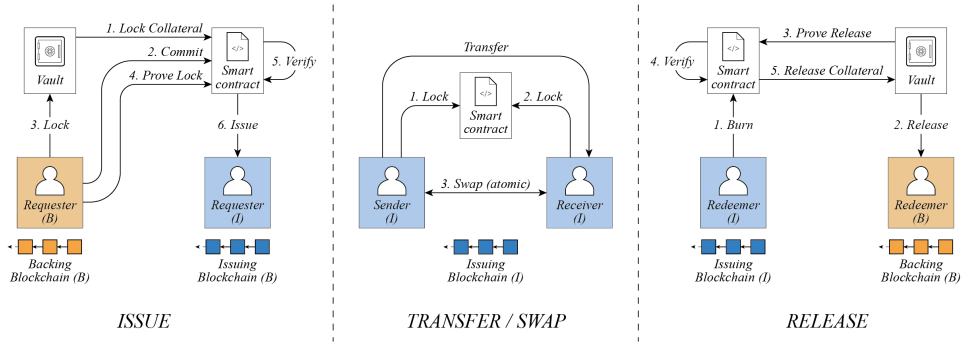


Figure 2.1: Overview of the different protocols in Xclaim.

b. Besides, Xclaim defines two protocols *Transfer* and *Swap*, outlining the transfer of $i(b)$ among participants on I and its exchange for i, respectively, with which we do not concern ourselves here. A diagram from the original Xclaim paper showcasing the different stages in these protocols can be seen on Fig. 2.1.

Issue and *Redeem* are defined as follows:

Protocol: Issue. Alice (requester) locks units of b with a vault on B to create $i(b)$ on I:

1. *Setup.* The vault registers with the iSC and locks i_{col} units of collateral, where it must hold that

$$i_{col} \geq b_{lock} \cdot r_{col} \cdot \varepsilon_{(i,b)} \geq b_{lock} \tag{2.1}$$

   where b_{lock} is the amount of funds that Alice will lock, r_{col} is an over-collateralisation factor in order to account for future exchange rate fluctuations, and $\varepsilon_{(i,b)}$ is the exchange rate at the time Alice commits to locking funds (next step).

   Alice verifies the iSC smart contract is available on chain I, i.e., the issuing blockchain, and both chooses a vault that satisfies Eq. (2.1) and learns its identity on B through the iSC.

2. *Commit.* Alice generates a new public/private key pair on I and commits to locking funds b_{lock} with the vault by providing a small amount of collateral to the iSC. This is in order to avoid *griefing* attacks in which Alice repeatedly commits to issuing with the vault without actually doing so. The collateral is refunded to Alice when she issues the funds.

   In future issue requests, the vault will only be able to issue $i(b) = \frac{i_{col}}{r_{col} \cdot \varepsilon_{(i,b)}} - b_{lock}$ and the amount of collateral backing b_{lock} is said to be

blocked. If Alice fails to complete *Issue*, however, this collateral becomes *free* again.

3. *Lock.* Alice locks funds b_{lock} with the vault on B in a publicly verifiable manner, i.e., by sending b_{lock} to the vault. As part of locking these funds with the vault, Alice also specifies where the to-be-generated $i(b)_{issue}$ should be sent, i.e., Alice associates her public key on I with the transfer of b_{lock} to the vault.

4. *Create.* Alice proves to the iSC that the funds were locked by providing an inclusion proof for her transaction to the chain relay. The iSC verifies that the transaction is correct and consensus has been reached on the given block, then creates and sends $i(b)_{issue}$ to Alice's public key in the transaction such that $|i(b)_{issue}| = |b_{lock}|$.

Protocol: Redeem. Dave (redeemer) locks $i(b)$ with the iSC on I to receive b from a vault on B; the $i(b)$ are then destroyed:

1. *Setup.* Dave creates a new public/private key pair on B and chooses a vault that can release the amount of funds $b_{release}$ he wishes to redeem, i.e. holds b such that $b_{vault} \geq b_{release}$, through the iSC.

2. *Lock.* Next, Dave locks $i(b)_{burn}$ with the iSC on I and requests their redemption. Thereby, Dave also specifies the vault he has chosen to release his funds and his public key on B as the target for the redeem.

3. *Release.* The vault witnesses the locking and redemption request of $i(b)_{burn}$ on I and releases funds $b_{release}$ to Dave's specified public key on B, such that $|b_{release}| = |i(b)_{burn}|$.

4. *Burn.* Finally, the vault confirms with the iSC that $b_{release}$ was redeemed on B by providing an inclusion proof for the release transaction, and the iSC burns the locked $i(b)_{burn}$ on I. The amount of the vault's collateral backing $b_{release}$ (see *Issue* protocol) becomes free again.

   Alternatively, if the vault fails to confirm the release, it is slashed and Dave is compensated in i from the vault's collateral such that $|b_{release}| = |i_{slash}|$.

Based on Δ_{submit}, Δ^I and Δ^B, sensible timeouts for the different stages of these protocols are suggested. A formal specification of all protocols in XCLAIM can be found in [14, Section VI-A].

Finally, XCLAIM employs automatic liquidation to address exchange rate fluctuations outside of the over-collateralisation factor. That is, if a vault's observed collateral rate $r^*_{col} = \frac{i_{col}}{r_{col} \cdot \varepsilon_{(i,b)}}$ is critically close to the lower bound of 1.0, the iSC allows anyone to redeem their $i(b)$ for i at a markup until the vault's entire collateral has been sold.

2.2 ZCash

Zcash is an implementation of the Decentralized Anonymous Payment scheme Zerocash [15]. It builds on Bitcoin's [20] transparent payment scheme, adding a shielded payment scheme to it that leverages zero-knowledge succinct non-interactive arguments of knowledge (*zk-SNARKs*) to enable private payments.

Although Bitcoin's payment scheme is an integral part of Zcash (in fact, as noted by Kappos et al. [26], over 85% of all Zcash transactions up until January 2018 had taken place solely within Bitcoin's transparent payment scheme), this work focuses on bringing cross-chain interoperability to Zcash's shielded payment scheme. Xclaim already defines an interoperability framework for transparent payment schemes such as Bitcoin, hence the contributions in this work may be thought of as supplementary to Xclaim's design in order to obtain a framework that covers all types of transactions in Zcash.

Therefore, after a brief explanation on how these two payment schemes coexist and interact in Zcash, the rest of this work will focus on Zcash's shielded payment scheme.

Transactions in Zcash can contain transparent inputs, outputs, and scripts, which all work as in Bitcoin, and shielded *JoinSplit*, *Spend* and *Output descriptions*, which are the encodings of *JoinSplit*, *Spend* and *Output transfers* in a transaction. A transaction may combine transparent and shielded components such that value is moved from the *transparent value pool* to the *shielded value pool* or vice-versa. This process is referred to as *shielding* or *deshielding*, respectively.

JoinSplit transfers were introduced in the original implementation of Zcash and are still supported in Sapling for backwards compatibility. We choose not to support these and instead design Zclaim based on the Spend and Output transfers introduced in the Sapling upgrade, which offer a more flexible approach to shielded spending among other improvements not discussed here.

Spend and Output transfers are analogous to transparent inputs and outputs, respectively. Each Spend transfer spends a shielded *note*, and each Output transfer creates one. A note represents that a value v is spendable by the recipient who holds the *spending key* corresponding to the destination *shielded payment address*. A note's sender, recipient and value are never revealed.

To each note there is a cryptographically associated *note commitment*, which is added to the *note commitment tree* when the note is created. Only notes whose note commitment is in the note commitment tree can be spent.

When the note is spent, a unique *nullifier* associated with that note must be revealed and is then added to the *nullifier set*. It is infeasible to compute the nullifier without the spending key[1] corresponding to the recipient's shielded payment address. Only notes whose nullifier is not in the nullifier set can be spent.

The main premise of Zcash's shielded payment scheme is that when a note is spent, the spender only proves that its note commitment is in the note commitment tree, without revealing which one. Revealing its nullifier also does not give away its note commitment. This means that a spent note cannot be linked to the transaction in which it was created.

Spend and Output transfers both contain a *value commitment* to the value of the note being spent or created. Furthermore, they contain computationally sound zk-SNARKs proofs and signatures which prove that all of the following hold except with insignificant probability.

For Spend transfers:

- the spender knows a note whose note commitment is in the note commitment tree;
- the revealed value commitment was derived from the value in the note;
- the spender knows the proof authorising key (which is again derived from the spending key, see Section 2.2.2 for an overview of key components in Sapling and the relations between them) corresponding to the shielded payment address in the note;
- the revealed nullifier is computed correctly.

And for Output transfers:

- the revealed note commitment is computed correctly;
- the revealed value commitment was derived from the value in the note used to generate the note commitment.

All Spend and Output transfers in a transaction, along with any transparent inputs and outputs, are checked to balance by verifying that the sum of all value commitments and of all transparent values is equal to zero.

2.2.1 Commitment schemes

A commitment scheme is a function that takes an input and a random *commitment trapdoor*, and returns a *commitment* to the input such that:

- No information is revealed about it without the trapdoor ('hiding'),

[1] Technically, without the nullifier deriving key, which is derived from the spending key.

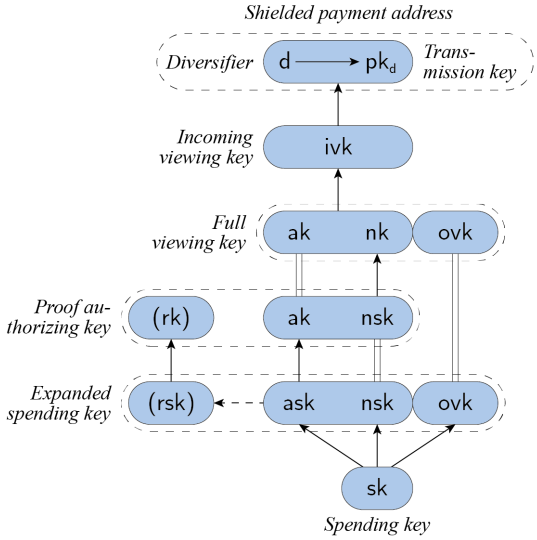


Figure 2.2: Sapling address components. Arrows signalise that a component is derived from another one. Double lines denote the same component.

- Given the trapdoor and input, the commitment can be verified to *open* to that input and no other ('binding').

Commitment schemes are used to hide the value in Output and Spend transfers and to generate note commitments. The precise definition can be found under [17, p. 4.1.7].

2.2.2 Addresses

A shielded payment address, which we may refer to simply as a *payment address*, consists of a *diversifier* d and a *transmission key* $\mathsf{pk_d}$. This address is ultimately derived from a spending key as depicted in Fig. 2.2.

As for the intermediate components, the *expanded spending key* is composed of a *spend authorising key* ask and a *nullifier private key* nsk, which are necessary to spend a note, and the outgoing viewing key ovk, which allows a user to decrypt notes spent by themselves. The *proof authorising key* allows devices that are computationally or memory-limited to delegate the generation of the zk-SNARK proof in Spend transfers to a third party. The key rsk is used to sign transactions and is a re-randomisation of the spend authorising key, while rk is the public key derived from it.

A *full viewing key* allows users to decrypt notes sent and received by them. Specifically, received notes can be decrypted using the *incoming viewing key* ivk derived from the *nullifier deriving key* nk and the (unnamed) ak compon-

ents of the full viewing key. Spent notes can then be identified by using the nullifier deriving key to derive the nullifiers of received notes and checking for their existence in the nullifier set. Together with the outgoing viewing key, this allows a user to decrypt all previous and future activity associated with the corresponding spending key.

See [17, Section 4.2.2] for types and derivation methods of the different key components.

2.2.3 Notes

Notes are similar to Bitcoin's Unspent Transaction Outputs or UTXOs, with the difference that by definition only unspent transaction outputs are stored in the UTXO set in order to be spent at a later point without needing to scan the entire blockchain. In Zcash, in contrast, all note commitments are stored in the note commitment tree, as it is not possible to tell an unspent note from a spent one.

A Sapling note $\mathbf{n}$ is defined as the tuple

$$(\mathsf{d} : \mathbb{B}^{[\ell_{\mathsf{d}}]},\ \mathsf{pk_d} : \mathbb{J},\ \mathsf{v} : \{0\,..\,2^{\ell_{\mathsf{value}}} - 1\},\ \mathsf{rcm} : \{0\,..\,2^{\ell_{\mathsf{scalar}}} - 1\})$$

where:

- d is the diversifier of the recipient's payment address.
- $\mathsf{pk_d}$ is the diversified transmission key of the recipient's payment address.
- v is an integer representing the value of the note in *zatoshi*, ZCash's smallest unit of currency. 1 zatoshi = 10^{-8} ZEC.
- rcm is a random commitment trapdoor used to derive the note commitment for this note.

The *note plaintext* $\mathbf{np}$ is the representation of a note in transactions, defined as

$$(\mathsf{leadByte} : \mathbb{B}^{\mathbb{Y}}, \mathsf{d} : \mathbb{B}^{[\ell_{\mathsf{d}}]}, \mathsf{v} : \{0\,..\,2^{\ell_{\mathsf{value}}} - 1\}, \underline{\mathsf{rcm}} : \mathbb{B}^{\mathbb{Y}^{32}}, \mathsf{memo} : \mathbb{B}^{\mathbb{Y}^{512}})$$

where $\mathsf{leadByte}$ is always 0x01 for Sapling notes, and memo may contain any additional information that the sender of the note wishes to transmit to the receiver.

The corresponding note commitment is calculated using rcm as the trapdoor and the concatenation of the remaining values as the input value.

2.2.4 Note commitment tree

The note commitment tree is a Merkle tree of fixed depth. Every node in this tree contains a hash computed from their children in the next layer, except for the outmost layer containing the note commitments. The index of a note's commitment at this layer is called its *note position*. When a note is created, its commitment is added to the outmost layer as a new leaf and thus the value in all of the leaf's ancestors including the root changes. A *positioned note* refers then to the note along with its position in the note commitment tree.

Every transaction has an input and an output *treestates*, which denote the state of the note commitment tree before and after adding any newly created notes. A block's input and an output *treestates* are defined as the output treestate of the last transaction in the preceding block and in that block, respectively. An *anchor* is the Merkle root of the note commitment tree in a given treestate.

Hash values are computed using a non-padded version of the SHA-256 hash function as defined in [27].

2.2.5 Spend and Output descriptions

A Spend transfer is encoded in transactions as a Spend description, which is defined as the tuple $(\mathsf{cv}, \mathsf{rt}, \mathsf{nf}, \mathsf{rk}, \pi_{\mathsf{ZKSpend}}, \mathtt{spendAuthSig})$, where:

- cv is the value commitment to the value of the input note.
- rt is an anchor for the output treestate of a previous block.
- nf is the note's nullifier.
- rk is a randomized validating key derived from the spend authorisation key that should be used to validate `spendAuthSig`.
- `spendAuthSig` is a signature over the SIGHASH transaction hash as defined in [28]. This signature proves knowledge of the spending key and guarantees that this Spend transfer cannot be replayed in other transactions.
- π_{ZKSpend} is a zk-SNARK proof for a Spend statement with primary input $(\mathsf{cv}, \mathsf{rt}, \mathsf{nf}, \mathsf{rk})$, as defined in [17, Section 4.15.2].

On the other hand, an Output description consists of $(\mathsf{cv}, \mathsf{cm}_u, \mathsf{epk}, \mathsf{C}^{\mathsf{enc}}, \mathsf{C}^{\mathsf{out}}, \pi_{\mathsf{ZKOutput}})$, where:

- cv is the value commitment to the value of the output note.
- cm_u is an integer representation of the note commitment for the output note.

- epk is an ephemeral key agreement public key, used to derive the key for encryption of the transmitted note ciphertext.
- $\mathsf{C}^{\mathsf{enc}}$ is a ciphertext component containing the note plaintext, encrypted with a key agreement scheme using the private key esk from which epk was derived and the diversifed transmission key $\mathsf{pk_d}$ of the recipient. The recipient can decrypt this component using his incoming viewing key ivk and epk.
- $\mathsf{C}^{\mathsf{out}}$ is a ciphertext component that allows the holder of an outgoing viewing key to recover esk and $\mathsf{pk_d}$, which can then be used to recover the note plaintext from $\mathsf{C}^{\mathsf{enc}}$.
- π_{ZKOutput} is a zk-SNARK proof for an Output statement with primary input $(\mathsf{cv}, \mathsf{cm}_u, \mathsf{epk})$, as defined in [17, Section 4.15.3].

Observe that the ciphertext components $\mathsf{C}^{\mathsf{enc}}$ and $\mathsf{C}^{\mathsf{out}}$ are not used as inputs in the SNARK. In fact, the correct encoding of these components is not enforced in Zcash, it being noted that 'it is technically possible to replace $\mathsf{C}^{\mathsf{enc}}$ for a given note with a random (and undecryptable) dummy ciphertext, relying instead on out-of-band transmission of the note to the recipient' [17, p. 46]. Similarly, not encrypting $\mathsf{C}^{\mathsf{out}}$ to one's ovk 'is useful if the sender prefers to obtain forward secrecy of the payment information with respect to compromise of its own secrets' [17, p. 34].

This leads to the possibility of value being lost irredeemably in a shielded note that has not been encrypted to its receiver, if the note plaintext is lost or never reaches the receiver. As we shall see, this has been a major challenge in designing ZCLAIM, since ensuring that a transaction has been made is not as easy as verifying that there is a note commitment for a note with the desired properties in the note commitment tree. Instead, it is also necessary to ensure that the recipient of the note has *received* the note plaintext.

Chapter 3

Preliminaries

- Concepts are usually *emphasised* when first introduced.
- Most variables are typeset in sans-serif, though some may be in *cursive*. There is no strict distinction between the two.
- **Bold letters.** Bold letters are usually followed by a definition and indicate the name of something that may be referenced later.
- · This is a set of instructions.
  · This is the next step.

3.1 Notation

Much of the notation in this work is borrowed from the Zcash protocol specification [17, Section 2]. Definitions are reproduced here for ease of reading together with the rest of notation:

$\mathbb{B}$	the type of bit values, i.e. $\{0, 1\}$
$\mathbb{B}^{\mathbb{Y}}$	the type of byte values, i.e. $\{0\,..\,255\}$
$\mathbb{N}$	the type of nonnegative integers
$\mathbb{R}$	the type of real numbers
$\mathsf{x} : T$	variable x is of type T
$S \xrightarrow{\mathrm{R}} T$	the type of a randomised algorithm
$x \xleftarrow{\mathrm{R}} f(s)$	sampling a variable from the output of f applied to s, given $f : S \xrightarrow{\mathrm{R}} T$ and $s : S$
$f_x(y)$	$f(x, y)$
$T^{[\ell]}$	the set of sequences of ℓ elements of type T
"`string`"	the string '`string`' represented as a sequence of bytes in US-ASCII
$\{a\,..\,b\}$	the set or type of integers from a through b inclusive
$a \,\|\|\, b$	the concatenation of sequences a then b

$\mathbb{F}_n$	the finite field with n elements
$[k]P$	scalar multiplication in a group as defined in [17, Section 4.1.8]
$x\star$	bit-sequence representation of x, where x is a group element
&	bitwise AND
$\bot$	unavailable information or failure
$\top$	success

3.2 Sapling functions

Below is a list of Sapling functions along with their section numbers in the protocol specification, for reference:

$\mathsf{MerkleCRH}^{\mathsf{Sapling}}$	5.4.1.3
$\mathsf{DiversifyHash}$	5.4.1.6
Sym	5.4.3
$\mathsf{KA}^{\mathsf{Sapling}}$	5.4.4.3
$\mathsf{KDF}^{\mathsf{Sapling}}$	5.4.4.4
RedDSA	5.4.6
$\mathsf{RedJubjub}$	5.4.6
$\mathsf{WindowedPedersenCommit}$	5.4.7.2
$\mathsf{NoteCommit}^{\mathsf{Sapling}}$	5.4.7.2
$\mathsf{ValueCommit}$	5.4.7.3
$\mathsf{Extract}_{\mathbb{J}^{(r)}}$	5.4.8.4
$\mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}$	5.4.8.5

3.3 Sapling constants

The following Sapling constants that appear in this document are defined in [17, Section 5.3] and are also reproduced here for convenience:

$\mathsf{MerkleDepth}^{\mathsf{Sapling}} := 32$

$\ell_{\mathsf{value}} := 64$

$\ell_{\mathsf{MerkleSapling}} := 255$

$\ell_{\mathsf{d}} := 88$

$\ell_{\mathsf{ivk}} := 251$

$\ell_{\mathsf{scalar}} := 252$

The constants $r_{\mathbb{J}}$, $h_{\mathbb{J}}$, $\ell_{\mathbb{J}}$, variables $\mathbb{J}$, $\mathbb{J}^{(r)}$, $\mathcal{O}_{\mathbb{J}}$ and the function $\mathsf{repr}_{\mathbb{J}}$ are defined in the context of the Jubjub curve in [17, Section 5.4.8.3].

Chapter 4

ZCLAIM Overview

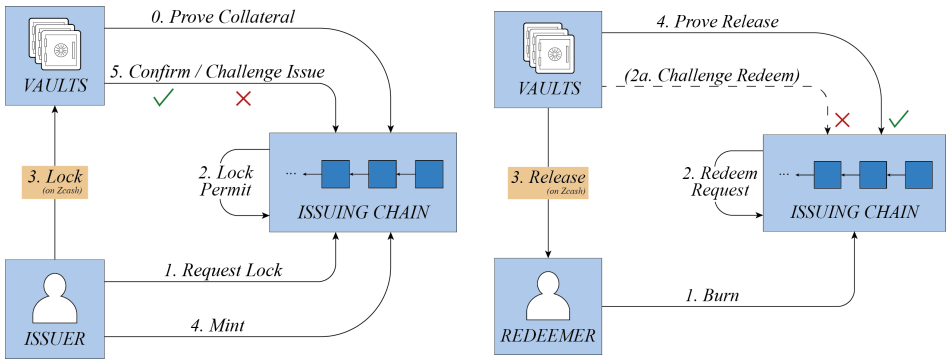
We introduce ZCLAIM, an adaptation of XCLAIM to the privacy coin Zcash. ZCLAIM adapts and expands on the ideas of XCLAIM such as to facilitate the transfer of value from a blockchain that is not publicly auditable to a smart-contract capable blockchain supporting the required cryptographic functions.

To this end, we assume an implementation of Zcash's Sapling upgrade according to the protocol specification [17] on the issuing chain and introduce new transfer types along with their accompanying zk-SNARKs to facilitate interoperability. These transfers fit into Zcash's shielded payment scheme and enable the issuing and redeeming of value. Furthermore, we discuss the smart-contract logic necessary to carry out the issue and redeem protocols of XCLAIM in this context.

4.1 Transfers and transactions

Along with the Spend and Output transfers of Sapling, the issuing chain offers Mint and Burn transfers. A Mint transfer is a transfer creating value of the issued currency, i.e. increasing the circulating supply. A Burn transfer takes the burnt amount out of circulation. Transactions in the issuing chain may contain transparent inputs, outputs, and scripts, shielded Spend and Output transfers, and a number of either Mint or Burn transfers.

A zk-SNARK in Mint transfers proves that a shielded note has been sent to the vault on Zcash and that the minted value corresponds to the locked value. In Burn transfers, redeemers create a note they wish to receive and encrypt it to the vault, while publishing its note commitment. The zk-SNARK therein allows redeemers to show that the note has the correct value. Once the vault proves that the note commitment has been added to Zcash's note



(a) Issue (b) Redeem

Figure 4.1: Simplified diagrams of ZCLAIM protocols.

commitment tree, the transaction containing the Burn transfer is included in the blockchain.

We refer to a transaction containing a Mint transfer as a mint transaction, and one containing a Burn transfer as a burn transaction. Transactions may contain at most either one Mint transfer or one Burn transfer, never more.

4.2 Protocols

ZCLAIM define two protocols, Issue and Redeem, which are adapted from XCLAIM to address the challenges described in Section 5.4. We present here a high-level overview of both protocols. Diagrams for both protocols depicting the simplified sequences of steps described in this section can be seen on Fig. 4.1.

4.2.1 Issuing

The Issue protocol allows the creation of ICZ by locking ZEC with a vault. It involves the following steps:

Protocol: Issue. Alice (issuer) locks funds with vault V on Zcash to create units of ICZ on I.

1. *Setup.* V registers with the vault registry on I and locks ICN_{col} units of collateral, where it must hold that

$$ICN_{col} \geq \mathsf{v}_{\mathsf{max}} \cdot (1 - f) \cdot \sigma_{std} \cdot xr_{cap} \tag{4.1}$$

where $\mathsf{v}_{\mathsf{max}}$ is the maximum amount of funds that can be locked or burned per request, f is the ZCLAIM transaction fee and σ_{std} is the standard collateralisation rate, all of which are defined in Section 5.8,

and xr_{cap} is the exchange rate at the time the vault provides a proof of capacity as explained next.

In order to register as a vault, V must also provide a diversified payment address on Zcash, to which issuers will send their funds, to the vault registry. V further submits a *proof of capacity* to the vault registry in which they prove that Eq. (4.1) holds, hence becoming available for lock requests.

2. *Commit.* Alice makes a request to the issuing chain to lock her funds with vault V with diversified payment address $(\mathsf{d}, \mathsf{pk_d})_{\mathrm{V}}$. As part of this request, Alice locks a small amount of ICN ICN_w, her *warranty collateral*, to prevent griefing attacks and compensate for V's opportunity cost in case Alice does not follow through with her request.

3. *Lock permit.* Subsequently, a *lock permit* is created on the issuing chain granting permission to Alice to lock her funds with V. This permit contains a cryptographic nonce n_{lock} that Alice must include in the transaction locking the funds.

   If Alice fails to execute step 4 within Δ_{mint}, her warranty collateral ICN_w is transferred to V.

4. *Lock.* Alice creates a shielded transaction on Zcash, sending ZEC_{lock} to V. Alice uses the nonce in the lock request to derive the note commitment trapdoor of the output note $\mathbf{n}$ addressed to V in this transaction.

5. *Create.* Alice makes a request to issue ICZ_{create} to a shielded address on I. To this end, Alice provides a note commitment $cm_{\mathbf{n}}$ and a Merkle proof to the root of the note commitment tree in a block header, and further proves in zero knowledge that:

   - she knows a note $\mathbf{n}$ with note commitment $cm_{\mathbf{n}}$, recipient $(\mathsf{d}, \mathsf{pk_d})_{\mathrm{V}}$ and value ZEC_{lock}
   - ICZ_{create} is equal to ZEC_{lock} minus fees and $ZEC_{lock} \leq \mathsf{v_{max}}$
   - the trapdoor used to generate $cm_{\mathbf{n}}$ was derived from n_{lock}

   Furthermore, Alice publishes the note ciphertext C^{V} of the note $\mathbf{n}$ encrypted to V. The transaction T_{mint} creating the funds ICZ_{create} is not included in I until V confirms it. If V fails to do so within $\Delta_{confirmIssue}$, the same amount of collateral ICN_w is deducted from V's collateral and transferred to Alice, and T_{mint} is included in I.

6. *Confirm/Challenge.* V decrypts C^{V} and verifies whether the resulting note has note commitment $cm_{\mathbf{n}}$. If this is the case, V sends a confirmation and the issuing process is complete.

   On the other hand, if V finds that they cannot properly decrypt C^{V}, they may challenge the transaction by revealing the shared secret used

in the encryption. It can then be verified on chain that the encryption was erroneous, in which case T_{mint} is discarded and Alice loses the funds ZEC_{lock} and ICN_w.

4.2.2 Redeeming

The Redeem protocol allows the redeeming of ICZ for ZEC, which involves destroying or *burning* ICZ on I and the release of ZEC by a vault. This is accomplished in the following sequence of steps:

Protocol: Redeem. Dave (redeemer) burns ICZ on I and obtains ZEC from vault V.

0. *Setup.* V is available to redeem, i.e. has not provided a *proof of insolvency* to the vault registry since they last participated in an Issue procedure.
1. *Burn.* Dave makes a request to burn funds ICZ_{burn} on I by locking ICN_w as warranty collateral and submitting a transaction T_{burn}.
2. *Redeem request.* In this transaction, Dave specifies that he would like to redeem ZEC from V in a note $\mathbf{n}$ with note commitment $cm_{\mathbf{n}}$, and proves in zero knowledge that:
   - he knows a note $\mathbf{n}$ with note commitment $cm_{\mathbf{n}}$ and value $ZEC_{release}$
   - $ZEC_{release}$ is equal to ICZ_{burn} minus fees and $ICZ_{burn} \leq \mathsf{v}_{\mathsf{max}}$

   In order to transmit the note values to V, Alice publishes the note ciphertext C^{V} of the note $\mathbf{n}$ encrypted to V.

   T_{burn} is not included in I until V confirms the release of $ZEC_{release}$. If V' fails to do so within $\Delta_{confirmRedeem}$, ICN_w is deducted from V's collateral and transferred to Dave, and T_{burn} is discarded.
3. *Release.* V releases $ZEC_{release}$ to Dave by creating a note $\mathbf{n}$ with note commitment $cm_{\mathbf{n}}$.
4. *Confirm/Challenge.* V waits until the relay system signals that consensus has been reached on the block in which $\mathbf{n}$ was created and then submits an inclusion proof for this note. T_{burn} is then included in I.

   However, if upon decryption of the note ciphertext C^{V} provided by Alice in step 2, V finds that the resulting plaintext does not correspond to a note with note commitment $cm_{\mathbf{n}}$, they may challenge the transaction by revealing the shared secret used in the encryption. It can then be verified on chain that the encryption was erroneous, in which case T_{burn} is discarded and Dave's warranty collateral is transferred to V. In this case, V does not execute step 3.

Chapter 5

Protocol

In this section, we present the core of this work, the ZCLAIM protocol.

We first introduce the basic setup, design goals and challenges. Next, we present the building stones of ZCLAIM: the necessary components on the issuing chain, operations and states to represent the Issue and Redeem protocols and data structures. Finally, we provide a specification of these operations, particularly of those that interact heavily with Zcash's shielded payment scheme and lay out policies for compensation and punishments.

5.1 Setup

In ZCLAIM, Zcash assumes the fixed role of the backing chain, whereas the issuing chain may be any smart-contract capable blockchain that meets the requirements in Section 5.2. Zcash provides all operations required for the backing chain as defined in the XCLAIM paper, as already pointed out in the same.

We adopt the blockchain model of XCLAIM as described in Section 2.1.2. Specifically, we make use of the security parameters k^Z, k^I, Δ^Z and Δ^I described in Section 2.1. Concrete values for these parameters are to be determined in future work, but we may take deposit processing times on popular cryptocurrency exchanges as a guideline, which are commonly 24 confirmations or 30 minutes [29, 30, 31] for Zcash transactions[1]. We remark that if the issuing chain offers deterministic finality, Δ^I may be much smaller than Δ^Z.

ZCLAIM defines the same actors as in XCLAIM, with the difference of the issuing chain I taking on the role of the iSC. This is to denote the fact that I may have been designed for the purpose of integrating with Zcash, and as

[1]The estimated delay to reach 24 confirmations is wrongly stated to be 60 minutes on these sources. This is likely due to an outdated block generation rate being used in the calculations, which was halved on December 11, 2019 as per the Blossom protocol upgrade [32]

such the components and operations therein would not be part of a smart contract, but integral components of the blockchain itself. Nevertheless, if Zclaim is implemented on a smart contract, all references to I can be understood to refer to the smart contract.

We set the backing currency as shielded Zcash (ZEC[2]). Moreover, we denote by ICN the issuing chain's native currency, and by ICZ the issued currency. Lastly, we assume that the issuing chain's native currency can also be shielded and deshielded, using either Zcash's or another shielded payment scheme.

5.2 Blockchain requirements

As mentioned above, the implementation of Zclaim can theoretically be carried out on any smart-contract capable blockchain, though it is very likely that doing so without native support for the cryptographic functions in Zcash would result in a prohibitively expensive protocol.

Block header verification One the one hand, there is a continuous cost associated with the verification of Zcash block headers by the relay system. This cost is carried by any blockchain attempting to build a relay to Zcash, regardless of support for shielded transactions. As such, the requirements to verify Zcash headers efficiently have been discussed previously in the context of integration with other projects, most notably Ethereum [33, 34, 35].

The consensus seems to be that the only strict requirement is native, i.e. optimised, support for the BLAKE2b compression function 'F' [36, Section 3.2] or, less preferably, for the full BLAKE2b hash function. This is because Zcash uses the Equihash [37] memory-hard proof-of-work algorithm, which requires 2^k iterations of the 'F' function to verify, where $k = 9$ in ZCash. As BLAKE2b is heavily optimised for 64-bit CPUs, performing this number of rounds on an unoptimised implementation for every new block, i.e. roughly every 2.5 minutes [17], is likely to prove too or at least unnecessarily expensive. This delay was halved in the protocol upgrade following Sapling, Blossom.

Sapling support Furthermore, in order to verify the zk-SNARKs in shielded transactions on I, support for elliptic curve arithmetic and pairings is required, in particular for the BLS12-381 pairing as defined in [17, Section 5.4.8.2] based on [38, 39] and the Jubjub curve used in Sapling [17, Section

[2]ZEC commonly denotes both shielded and transparent units of currency on Zcash. Since this work is only concerned with shielded transactions, we choose to redefine this symbol for simplicity.

5.4.8.3]. In practice, Sapling could be implemented used different curves and pairings, which would however incur a far larger implementation effort. We assume that operations on these curves and pairings are optimised such that the cost of zk-SNARK verification is similar to that in Zcash. Sapling also relies on the SHA-256 [27], BLAKE2b and BLAKE2s [40, 36] hash functions, hence we assume an efficient implementation for these is also provided. Prior to the Sapling upgrade, Zcash further used the SHA-256 compression function SHA256Compress as defined in [17, Section 5.4.1.1] and SHA-512, though these may not be necessary if only supporting Sapling notes.

These requirements being met, the implementation of Sapling on another blockchain or even on a smart contract is entirely possible. This is currently being done on Tezos [25, 41], and a number of adaptations of the Zerocash protocol already exist in smart contracts, for instance on Ethereum [42] and Quorum [43], as well as various general-purpose privacy protocols [44, 45] based on the technology developed for Zcash.

5.3 Design goals

We choose to deviate from the security properties of XCLAIM reproduced in Section 2.1.3, with the following rationale:

- **Auditability.** Naturally, auditability cannot be satisfied in ZCLAIM due to the nature of shielded transactions.
- **Consistency.** As reasoned in Section 6.2.5, consistency as defined in XCLAIM also cannot be guaranteed in ZCLAIM, as it is not possible to verify that backing funds are not being reused. We show that we are still able to guarantee that the issued funds are backed, here by the vaults' collateral instead of the backing currency.
- **Redeemability.** This property is satisfied in ZCLAIM through the same mechanisms as in XCLAIM.
- **Liveness.** We choose not to include liveness in our work, since arguably, even in XCLAIM, issuing does require a third party: a vault with sufficient collateral to cover for the funds being locked. Apart from this observation, ZCLAIM could be considered to satisfy liveness as it is technically possible to issue funds without interaction from the partaking vault. However, as explained later on, vaults should either provide confirmation or challenge lock transactions, in the absence of which they will be subjected to slashing. Thus issuing, if correctly executed, is in fact interactive.
- **Atomic Swaps.** Atomic swaps are considered to take place outside of ZCLAIM, i.e. they are regarded as functionality that is provided by the issuing chain and is not part of the ZCLAIM protocol.

Instead, we define the following desirable properties:

- **Soundness** The total amount of issued currency in circulation is equal to the total amount of ZEC obligations, i.e. the amount of ZEC for which vaults need to prove to be collateralised.
- **Coverage** ZEC obligations are backed by a proportional amount of the issuing chain's native currency according to the prevailing exchange rate.
- **Fairness** An honest participant following best practices will not incur any loss of funds as long as they can receive and broadcast transactions from and to chains Z and I.
- **Untraceability** An adversary will not be able to infer a user's identity through observation of their activity within the protocol.
- **Minimal modifications** We design Zclaim such as to diverge as little as possible from the Sapling protocol.

Soundness, coverage and fairness are formally defined and shown to hold in Section 6.1.2. Untraceability is covered in Section 6.3 and Section 6.2.1. As for the last property, the purpose is to minimise the implementation effort of Zclaim by making it possible to reuse code from existing libraries and clients, such as [46, 47, 48] for a Rust implementation. In particular, we:

- strive to create zk-SNARKs that can be constructed using the circuit components already implemented in Sapling,
- design Mint and Burn transfers such as to fit into Sapling's shielded payment scheme.

5.4 Challenges

Naturally, we face a number of challenges due to the private nature of the protocol. We classify these challenges under two categories: those related to interoperability, and those deriving from the integration of Zclaim and Sapling.

5.4.1 Interoperability challenges

Intending to interoperate with a blockchain that was designed for privacy is bound to pose some difficulties. The first and main challenge lies in verifying claims about shielded transactions without being able to 'see inside' of them, i.e. without requiring the sender to open the transaction hence revealing private data. Another issue is the question of whether this can be accomplished using only the block headers or if read access to the transac-

tion data is required. Lastly, the size of Zcash headers make it infeasible to store all headers on the issuing chain, which would be optimal for security.

Opacity of transactions XCLAIM relies on the public nature of transactions on the backing chain, which allow participants to prove statements about transactions by simply providing the data constituting a transaction to the iSC and showing that it was included in a block. This is much harder to accomplish in ZCLAIM, since transactions do not reveal information about shielded Output notes that have been created in them. Specifically, the receiver and value of the note are only proven to be valid in the zk-SNARK and cannot be extracted from transaction data.

This leaves two approaches to verify a statement of the form 'a note **n** of value v with recipient $(\mathsf{d}, \mathsf{pk_d})$ exists in the note commitment tree', which is the type of statement we need to prove in ZCLAIM.

The first and easiest approach is to reveal the note values and show that there exists a note commitment to this note in the note commitment tree. However, this is clearly not a viable approach for our use case, since it defeats **Untraceability** mainly by revealing the note value. The diversified payment address of the recipient can be re-randomised to be different for every transaction, but if if it is reused this also links several transactions with the same recipient.

The second and only viable approach then is to prove this statement in a zk-SNARK. This is done in the π_{ZKSpend} zk-SNARKs in Spend transfers, which however also reveal the nullifier of the note and require knowledge of the spending key associated with the diversified payment address of the recipient. We define custom zk-SNARKs later on in this chapter that allow the sender of a note to prove this statement without revealing the nullifier and without knowledge of the spending key.

Header-only transaction verification Since Zcash is implemented on top of Bitcoin, Bitcoin relays can be used as a reference point for the Zcash relay needed in ZCLAIM. We discuss the header fields relevant for both relays.

Both Bitcoin and Zcash headers include the root of a Merkle tree containing all transactions in the block, which can be used to prove that a given transaction was included in a block by constructing a Merkle path from the transaction to the Merkle root.

The Bitcoin UTXO set, compared in Section 2.2 to Sapling's note commitment tree[3], is a key-value database and hence cannot be stored efficiently in headers such as to provide access to its contents, as opposed to the note

[3]Zcash also keeps a UTXO set containing unspent transparent outputs, which are not used in ZCLAIM.

commitment tree. This means that in Bitcoin relays, users need to submit a transaction along with a Merkle path to the Merkle root in the containing block for verification.

Sapling, however, defines a field `hashFinalSaplingRoot` in block headers that represents the root of the note commitment tree in the output treestate of the block, i.e. after all note commitments created in transactions in this block have been added to the tree.

This represents an opportunity for interoperability not achievable in Bitcoin relays: it is possible to prove that a (spent or unspent) note exists without providing the transaction in which it was created. This allows for more efficient inclusion proofs, since the transaction data does not need to be submitted to the issuing chain. A typical shielded transaction with one Spend transfer and two Output transfers, one to the receiver and one containing the change, is 2 469 bytes in size. Hence by circumventing the need to provide the transaction in which a note was created, we reduce the size of inclusion proofs by this amount.

Transactions, however, contain more information than the note commitment tree, most notably the note plaintext encrypted to the receiver. In Zcash, note values may be transmitted to the receiver in- or out-of-band, i.e. the sender of a note is not required to encrypt the note plaintext to the receiver on-chain. Hence in order to guarantee that the receiver of a note is able to spend it, it is necessary to either verify that the note plaintext has been encrypted to them or to put in place a secure mechanism allowing them to challenge inclusion proofs by the sender if and only if the note plaintext has not been correctly transmitted.

The first option requires the sender to submit the transaction to the issuing chain and to prove in zero knowledge that the note ciphertext therein is equal to the note plaintext encrypted to the receiver. Apart from the larger proof size, this has the added drawback of needing to implement AEAD_CHACHA20_POLY1305 [49] verification in zero knowledge, the encryption scheme used in Zcash to encrypt note plaintexts.

As this goes against our stated **Minimal modifications** design goal, we choose the latter option and introduce a mechanism allowing the receiver to challenge inclusion proofs if the note plaintext has not been correctly encrypted to them, detailed later on in this chapter.

Size of chain verification proof Another concern is the fact that the relay system needs to store a number of past block headers in order to verify a valid chain. Currently, this number is linear in block chain length, meaning that all block headers since genesis need to be verified. At 1 487 bytes per header as per the specification and 1 056 169 Zcash blocks having been

mined at the time of writing, this amounts to over 1.46 Gigabytes of data, which is simply too large to store, even temporarily, on most existing blockchains. For comparison, Bitcoin block headers are 80 bytes in size. At the current block height of 658 845, the storage requirements for a full chain verification amount to a mere 50 MB.

This disparency in header size is largely due to the Equihash solution included in Zcash headers, which makes up for 1 344 bytes out of the total 1 487. Short of a change to Zcash's proof-of-work algorithm that would reduce the size of the solution data[4], the only alternative is a modification to the headers that would allow for non-linear size verification. This was indeed introduced in an upgrade posterior to Sapling [51] codenamed Heartwood, with an update to the data structures referenced by headers allowing for probabilistic verification logarithmic in block chain length as per the FlyClient protocol [52].

Taking advantage of the reduced proof size enabled by this update would imply implementing the FlyClient protocol on I. Due to the recency of this update, which was released as this work was well underway, we do not explore this approach, but note that it presents a realistic solution to the aforementioned problem.

5.4.2 Integration challenges

The transparency requirements of Xclaim clash with the privacy features of Zcash, which becomes apparent in a number of challenges when designing Zclaim. On the one hand, vaults learn the amounts that issuers lock with them and those that the vaults themselves release to redeemers, which represents a privacy concern. On the other hand, ensuring that vaults are collateralised properly constitutes a technical challenge, since the amounts that are being issued and redeemed through transactions with a given vault are private.

Information leaked to vaults Regardless of the hypothetical impenetrability of the protocol from a cryptography perspective, the vault system implies an unavoidable leakage of some amount of information. Even though vaults never learn the identity[5] of issuers/redeemers neither on Zcash nor on the issuing chain, they learn the amount that the latter is issuing or redeeming, respectively.

This can lead to de-anonymisation of the participant through a number of attack vectors, as discussed in Section 6.2.1.

[4]This issue has been discussed by the community with inconclusive results, see [50].

[5]a traceable address which can link the transaction with other ones

The straightforward strategy to counter this issue is to split the total amount being exchanged among several transactions, each involving a potentially different vault. We define such a strategy in Section 5.14.

Ensuring vault collateralisation Since only the vaults have knowledge of the total amount that is locked with them at any given time, they need to provide *proofs of balance* in order to show that they have enough collateral to cover for these funds when the exchange rate fluctuates. They also need to prove statements about their balance in order to become eligible for new lock requests or to be exempted from redeem requests. We call these statements *proofs of capacity* and *proofs of insolvency*, respectively.

Failure to present a proof of balance upon request within a certain timeframe results in slashing in the form of *liquidation*, which entails the sale of a fraction of the vault's collateral in exchange for issued currency.

5.5 Components

The functionality Zclaim requires on the issuing chain can be split into a number of components. The components presented in this section are non-exhaustive and a large amount of logic in the issuing chain is not presented as part of a component. Instead, they are introduced here because their function is clearly defined and they play a prominent role in the protocol.

5.5.1 Vault registry

The vault registry keeps a public list of all registered vaults and their status. Each vault has a shielded payment address (d, pk_d) and an amount of collateral associated with them. The collateral they keep in ICN on the parachain and thus the total amount of ZEC they are able to accept are public; however, the amount of ZEC that is currently locked with them is not. They periodically prove that this amount is properly backed by their collateral by submitting proofs of balance.

Besides, vaults can be available for deposits and redeems, independently of each other. The only requirement for a vault to be available for deposits is that it has enough free collateral on the parachain and has proven so by submitting a proof of capacity. Vaults are available for redeem requests by default. In order to be exempted from redeem requests, they need to provide a proof of insolvency, showing that the amount of ZEC locked with them is smaller than the maximum amount redeemable in a redeem request.

See Section 5.12 for details on balance statements.

5.5.2 Relay system

The relay system keeps track of the state of the backing chain. Specifically, it verifies and stores block headers, provides a mechanism to verify that consensus has been reached on a given block, similarly to how an SPV or light client works [53, 54, 18], and allows to verify that a note has been created on Zcash. This is accomplished by means of the following functionality.

Difficulty adjustment policy The relay system implements Zcash's difficulty adjustment algorithm, described in [17, Section 7.6.3], which is updated after every block. This is necessary in order to verify that the Equihash solution to subsequent blocks is valid.

Block header validation Given the adjusted difficulty based on previous blocks and a newly submitted block header, the relay system can verify that the Equihash solution in that header is valid. The cost of this verification is 2^k XORs and iterations of the BLAKE2b compression function 'F', where $k = 9$ in ZCash.

Validation is only successful if the Equihash solution is valid for the current difficulty and its pre-image is equal to the header fields defined in [17, Section 7.6.1].

Chain validation The relay system can verify that a sequence of blocks $(\mathcal{B}_0, \mathcal{B}_1, ..., \mathcal{B}_n)$ constitute a valid chain by verifying that:

- $\mathcal{B}_0$ is the genesis block, which is stored on the relay system as a parameter.
- For each consecutive block $\mathcal{B}_{i+1}$, the hash of the previous block `hashPrevBlock` in the header is derived from the header of block $\mathcal{B}_i$ and the Equihash solution for block $i + 1$ is valid as described above.

Consensus verification Zcash uses Nakamoto consensus, which dictates that consensus participants agree on the chain with the most accumulated work [20]. This depends on the difficulty adjustment and the number of blocks of a given chain. Nakamoto consensus is probabilistic, which means that consensus is technically never fully reached. Hence the relay system relies on security parameter k and only deems blocks at depth h in a valid chain, where $h \geq k$, to have reached consensus. We say that such blocks, and the transactions therein, have been *confirmed* by the relay system.

Note commitment verification Given a note commitment hash cm_u and a Merkle path from cm_u to the `hashFinalSaplingRoot` field in the block

header of some block $\mathcal{B}$, the relay system verifies that the note commitment is valid if the Merkle path is correct and block $\mathcal{B}$ has been confirmed.

This makes it possible to verify that a note has been created on Zcash. On the other hand, the relay system offers no functionality to verify whether a given note has been spent, which is not required in the context of Zclaim.

Block headers are submitted to the relay system by *relayers*, which are third-party actors responsible for keeping the latter up-to-date. These actors may be economically incentivised in order to discourage dishonest behaviour, though other protocol actors will generally take on the role of relayers if there is no or low additional cost associated with doing so and it is in their interest to guarantee the proper functioning of the protocol. Vaults, for example, are in such a position: they need to run full nodes of both chains anyway, and the financial damage they may incur from an attack on the relay system is far greater than the cost of feeding block headers to the relay system themselves.

Finally, as long as there is one honest relayer, the cost of attacking the relay system is the same as that of running a 51% attack on Zcash, since block headers are verified based on proof of work and not on the number of relayers submitting them. See Section 6.2.2 for more details.

5.5.3 Exchange rate oracle

The exchange rate oracle $\mathcal{O}_{\mathsf{xr}}$ provides an exchange rate that reflects the prevailing market value of 1 ZEC in ICN. This value xr must be of the form $\mathsf{xr}_n/\mathsf{xr}_G$, where $\mathsf{xr}_G = 2^{\ell_{\mathsf{xr}_G}}$ is a fixed value representing the *exchange rate granularity*. On the other hand, $\mathsf{xr}_n : 0..2^{\ell_{\mathsf{xr}_n}}$ varies to reflect the exchange rate as obtained from external feeds. The design of the exchange rate oracle falls outside the scope of this work, but a brief discussion on exchange rate sourcing and security concerns can be found in Section 6.2.3.

5.6 Operations

We define abstract operations and states for the Issue and Redeem protocols based on the high-level overview in Section 4.2, which we will use later on to provide a complete specification of Zclaim. States are only explicitly referred to in this and the next section in order to provide a comprehensive picture of the protocols.

An operation always results in a transaction being submitted to either ZCash or the issuing chain. In the case of the issuing chain, this may be a monetary transaction or a change in state. For simplicity, we omit non-monetary transactions and warranty collateral transactions, both of which only take

place on the issuing chain. Each operation leads to a particular state of the protocol, in which usually the counterparty is required to submit an appropriate transaction within a certain delay. To each state there is an implicit associated set of legal operations for each party. Each vault always is in one Issue state and one Redeem state except upon registration, whereas issuers and redeemers are only assigned a state for the duration of an Issue or Redeem *procedure*, i.e. an instance of the Issue or Redeem protocols.

We denote by T^A_{op} the transaction on blockchain A resulting from the successful execution of the operation op. A may be any of Zcash, denoted by Z, or the issuing chain, denoted by I. Certain transactions on I require confirmation from the other party involved in the Issue or Redeem procedure and may be discarded if this confirmation is not provided within a certain delay. We denote a transaction that is pending confirmation as (T^I_{op}), and one that is discarded as $\cancel{(T^I_{op})}$.

An operation that results in transaction T_{op}, can be performed in state StartingState and leads to state OutputState is denoted by $op \rightarrow T^A_{op}$ [StartingState $\rightarrow$ OutputState]. Operations with no output transaction imply a non-monetary and/or a warranty collateral transaction on I. We classify operations as being performed by either an issuer, a redeemer or a vault.

Operations performed by the issuer:

- *requestLock* [VaultAcceptingIssue $\rightarrow$ AwaitingMint] requests permission to lock funds with a vault. The issuer locks ICN_w as collateral.
- *lock* $\rightarrow T^Z_{lock}$ [AwaitingMint $\rightarrow$ AwaitingMint] locks ZEC with a vault.
- *mint* $\rightarrow (T^I_{mint})$ [AwaitingMint $\rightarrow$ AwaitingIssueConfirm] allows the issuer to mint a hidden amount of ICZ on I.

  Transaction T^I_{mint} is only included in I if state IssueSuccess is reached.

Operations performed by the redeemer:

- *burn* $\rightarrow (T^I_{burn})$ [VaultAcceptingRedeem $\rightarrow$ AwaitingRedeemConfirm] burns a hidden amount of ICZ such that they can be redeemed. The redeemer locks ICN_w as collateral.

  Transaction T^I_{burn} is only included in I if state RedeemSuccess is reached.

Operations performed by the vault:

- *challengeIssue* [AwaitingIssueConfirm $\rightarrow$ IssueChallenged] allows a vault to prove that T^Z_{lock} has not been correctly encrypted to them in (T^I_{mint}).
- *challengeRedeem* [AwaitingRedeemConfirm $\rightarrow$ RedeemChallenged] allows a vault to prove that $T^Z_{release}$ has not been correctly encrypted to them in (T^I_{burn}).

- *release* $\rightarrow T^{Z}_{release}$ [AwaitingRedeemConfirm $\rightarrow$ AwaitingRedeemConfirm] releases locked funds to a redeemer.
- *confirmIssue* [AwaitingIssueConfirm $\rightarrow$ IssueSuccess] allows a vault to confirm that they have received T^{Z}_{lock}, confirming the pending (T^{I}_{mint}) transaction.
- *confirmRedeem* [AwaitingRedeemConfirm $\rightarrow$ RedeemSuccess] proves that the vault has released funds in $T^{Z}_{release}$, confirming the pending (T^{I}_{burn}) transaction.
- *submitPOB* [VaultRegistered|VaultAcceptingIssue|VaultNotIssuing $\rightarrow$ VaultNotIssuing] submits a proof of balance showing that the vault has enough collateral to cover for the ZEC obligations associated with them.
- *submitPOC* [VaultRegistered|VaultAcceptingIssue|VaultNotIssuing $\rightarrow$ VaultAcceptingIssue] submits a proof of capacity showing that the vault has enough collateral to accept more funds.
- *submitPOI* [VaultNotRedeeming|VaultAcceptingRedeem $\rightarrow$ VaultNotRedeeming] submits a proof of insolvency showing that vault does not have enough ZEC obligations to offer a redeem.

Using these operations, the Issue and Redeem protocols are summarised in pseudocode in Algorithms 1 and 2, respectively. Warranty collateral transfers and slashing are omitted for simplicity.

Algorithm 1 Issue

Require: vault has enough collateral
1: vault executes *submitPOC*
2: **if** issuer executes *requestLock* and receives a lock permit **then**
3: issuer executes $lock^{\rightarrow T^{Z}_{lock}}$
4: **if** issuer executes $mint^{\rightarrow (T^{I}_{mint})}$ within Δ_{mint} **then**
5: **if** vault executes *challengeIssue* within $\Delta_{confirmIssue}$ **then**
6: $\rightarrow \cancel{T^{I}_{mint}}$
7: **else**
8: vault may execute *confirmIssue* within $\Delta_{confirmIssue}$
9: $\rightarrow T^{I}_{mint}$
10: **end if**
11: **end if**
12: **end if**

Algorithm 2 Redeem

Require: vault has not called *submitPOI* since last issuing
1: redeemer executes $burn^{\rightarrow (T^{I}_{burn})}$
2: **if** vault executes *challengeRedeem* within $\Delta_{confirmRedeem}$ **then**
3: $\rightarrow \cancel{T^{I}_{burn}}$
4: **else**
5: vault executes $release^{\rightarrow T^{Z}_{release}}$
6: **if** vault executes *confirmRedeem* within $\Delta_{confirmRedeem}$ **then**
7: $\rightarrow T^{I}_{burn}$
8: **else**
9: $\rightarrow \cancel{T^{I}_{burn}}$
10: **end if**
11: **end if**

5.7 States

The following states are part of the Issue protocol. The transactions and changes in state resulting from reaching these states are specified hereinafter.

- VaultNotIssuing: The vault becomes unavailable for lock requests.
- VaultAcceptingIssue: The vault becomes available for lock requests.
- AwaitingMint: The issuer must call *mint* within Δ_{mint}. If they fail to do so, the protocol switches to state MintTimeout.
- MintTimeout: The issuer's warranty collateral is transferred to the vault and the protocol returns to state VaultAcceptingIssue.
- AwaitingIssueConfirm: The vault must execute *confirmIssue* within $\Delta_{confirmIssue}$. If they fail to do so, the protocol switches to state IssueConfirmTimeout.
- IssueChallenged $\rightarrow$ $(\cancel{T^I_{mint}})$: The issuer's warranty collateral is transferred to the vault and the protocol returns to state VaultAcceptingIssue.
- IssueConfirmTimeout $\rightarrow T^I_{mint}$: ICN_w is transferred from the vault's collateral to the issuer, and the issuer's warranty collateral is returned to the same. The protocol switches to state VaultNotIssuing.

  If the vault is in redeem state VaultNotRedeeming, the Redeem protocol switches to VaultAcceptingRedeem.
- IssueSuccess $\rightarrow T^I_{mint}$: ICN_w is returned to the issuer and the protocol returns to state VaultAcceptingIssue.

  If the vault is in redeem state VaultNotRedeeming, the Redeem protocol switches to VaultAcceptingRedeem.

Similarly, the following states appertain to the Redeem protocol.

- VaultNotRedeeming: The vault becomes unavailable to redeem.
- VaultAcceptingRedeem: The vault becomes available to redeem.
- AwaitingRedeemConfirm: The vault must call *confirmRedeem* within $\Delta_{confirmRedeem}$. If they fail to do so, the protocol switches to state RedeemConfirmTimeout.
- RedeemConfirmTimeout $\rightarrow$ $(\cancel{T^I_{burn}})$: ICN_w is transferred from the vault's collateral to the redeemer. The protocol switches to state VaultAcceptingRedeem.
- RedeemChallenged $\rightarrow$ $(\cancel{T^I_{burn}})$: The redeemer's warranty collateral ICN_w is transferred to the vault and the protocol returns to state VaultAcceptingRedeem.

- RedeemSuccess $\rightarrow T^{I}_{burn}$: ICN_w is returned to the redeemer and the protocol returns to state VaultAcceptingRedeem.

The state VaultRegistered is the state of a vault immediately upon registration. In this state, the vault is neither available for Issue nor for Redeem requests.

State diagrams for both protocols, depicting the relation between operations defined in the previous section and the states introduced in this section, can be found on Fig. 5.1.

5.8 Constants

The following constants are defined as presented in this section. Specific values should be chosen following testing and according to implementation constraints.

- σ_{std} is the standard collateralisation ratio.

  Vaults periodically provide POBs showing that their collateralisation ratio, i.e. the ratio of ZEC obligations to locked collateral according to the exchange rate, is above σ_{std}. Formally, for any vault V:

  · Let $\mathsf{v_m}_i$ be the value of the i-th mint transaction in which funds were locked with V, where $0 \leq i \leq n$
  · Let $\mathsf{v_b}_j$ be the value of the j-th burn transaction in which funds were released by V, where $0 \leq j \leq m$
  · Let xr_V be an exchange rate chosen by the vault, where $\mathsf{xr}_V \leq \mathsf{xr}_{\mathcal{O}}$, the latest exchange rate provided by the oracle.
  · Let coll be the vault's locked collateral.

  Then, a proof of balance allows a vault to prove in zero knowledge that

$$(\sum_{i=1}^{n} \mathsf{v_m}_i - \sum_{j=1}^{m} \mathsf{v_b}_j \cdot \mathsf{xr}_V \cdot \sigma_{std} \leq \mathsf{coll} \tag{5.1}$$

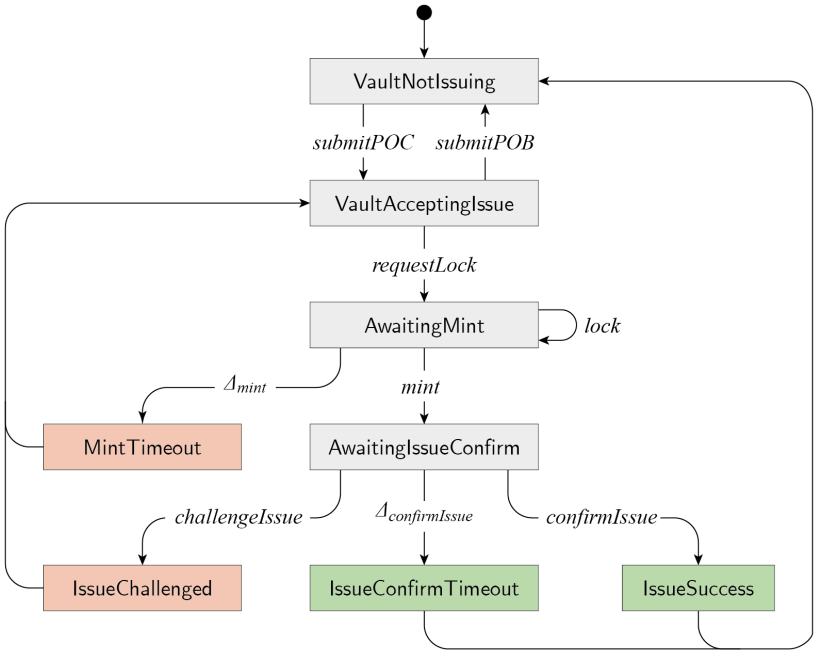
  holds.

  Similarly, in proofs of capacity vaults prove that this inequality holds for their future balance after a lock transaction, regardless of its value. See Section 5.12 for more details.

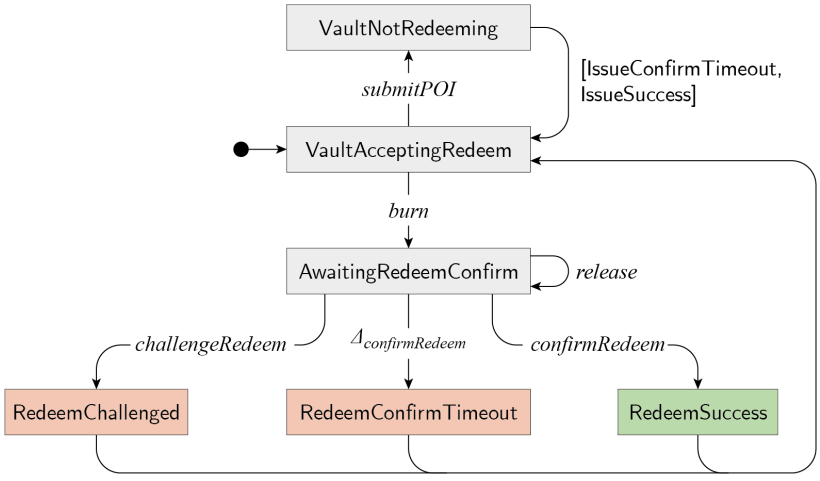
  σ_{std} must be of the form $\frac{p}{q}$, where $p, q \in \mathbb{N}$ and q is a power of 2, in order to make computations in zk-SNARKs in which σ_{std} is used more efficient.

  A suitable value may be $\sigma_{std} = \frac{3}{2} = 1.5$.

  This value does not need to be a constant but can instead be adjusted on a per-participant basis, where vaults with a trustworthy track



(a) Issue



(b) Redeem

Figure 5.1: State diagrams for the Issue and Redeem protocols. States colored green represent success, red ones failure.

record can move up in a tiered system, in order to minimise the opportunity cost due to over-collateralisation [55].

- σ_{min} is the minimum collateralisation ratio.

  Keeping in mind that the amount of ZEC obligations associated with a vault, $ZEC^{obl} = \sum_{i=1}^{n} v_{l,i} - \sum_{j=1}^{m} v_{r,j}$, is hidden, we can say that vaults provide an upper bound ZEC^{obl}_{max} for this value in Eq. (5.1), where $ZEC^{obl}_{\mathsf{max}} = \frac{coll}{xr_V \cdot \sigma_{std}}$. We now define a vault's observed collateralisation ratio $\sigma_V = \frac{coll}{xr_t \cdot ZEC^{obl}_{\mathsf{max}}}$ as the resulting lower bound on the collateralisation ratio assuming $ZEC^{obl} = ZEC^{obl}_{\mathsf{max}}$, where xr_t is the latest exchange rate provided by the oracle.

  In order to ensure that all issued funds are properly collateralised, σ_V can never fall below 1, which may happen if the exchange rate increases by a certain margin. Hence we define the minimum collateralisation ratio σ_{min}, where $1 < \sigma_{min} < \sigma_{std}$, as the liquidation trigger, i.e. liquidation is triggered on vault V whenever $\sigma_V \leq \sigma_{min}$. See Section 5.15 for more details on liquidation.

  For example, $\sigma_{min} = \frac{9}{8} = 1.125$.

- $\mathsf{v}_{\mathsf{max}}$ is the maximum value of an issue or redeem transaction in zatoshis, Zcash's smallest unit of currency.

  Specifically, this is the maximum amount of funds that can be locked or burned per transaction, i.e. prior to the subtraction of fees. This limit exists in order to protect the user's privacy, as otherwise amounts in the upper tail of the distribution of Zcash transaction values are more easily identifiable even if the splitting strategy described in Section 5.14 is employed.

  As argued in the aforementioned section, this value is a power of 2.

- $\mathsf{v}_{\mathsf{min}}$ is the minimum value of an issue or redeem transaction in zatoshis, i.e. the minimum amount of funds that can be locked or burned per transaction.

  This lower bound is defined such as to limit the number of transactions resulting from the splitting strategy. Besides, sending very small amounts through Zclaim is economically unprofitable as the incurred fees grow larger in proportion to the amount being transferred as the latter decreases, due to the rounding error when computing fees paid to vaults and to transaction fees on Zcash and the issuing chain.

  This value is also a power of 2 as per the splitting strategy.

- $f = \frac{f_n}{f_G}$, where $f_n \in \mathbb{N}$ and $f_G = 2^{\ell_f}$, represents the fraction of a transaction that goes towards the payment of fees to the vault involved.

The fee granularity f_G must be a power of two in order to facilitate efficient multiplication in zk-SNARKs. ZCLAIM's fee policy is described in detail in Section 5.13.

For example, $f = \frac{1}{256}$, which is along the lines of CBA issuing fees in existing protocols [56, 57].

- Δ_{mint} is the mint timeout.

  This is the delay from inclusion in I of a lock request to the moment it is discarded if no associated Mint transfer is submitted.

  The value Δ_{mint} is lower bounded by $\Delta^I + \Delta^Z + \Delta_{submit}$, i.e. the time the issuer must wait until their lock request and subsequent lock transaction are securely included in I and Zcash, respectively, and the lock transaction can be confirmed by the relay system.

- $\Delta_{confirmIssue}$ is the issue confirmation timeout.

  This is the delay within which a vault must confirm the reception of the note in an issuer's lock transfer after the latter has submitted a mint transaction. At the same time, it is the delay within which they may challenge the issuer's mint transaction if the encryption of the note was erroneous.

  $\Delta_{confirmIsssue}$ may be quite small, as the vault does not need to wait until a mint transaction is securely included in I to provide confirmation.

- $\Delta_{confirmRedeem}$ is the redeem confirmation timeout.

  This is the delay within which a vault must confirm the release of funds after a redeemer has submitted a burn transaction on I. At the same time, it is the delay within which they may challenge the redeemer's burn transaction if the encryption of the requested note was erroneous.

  $\Delta_{confirmRedeem}$ is lower bounded by $\Delta^I + \Delta^Z + \Delta_{submit}$, i.e. the time the vault must wait until the burn transaction and their release transaction are securely included in I and Zcash, respectively, and the release transaction can be confirmed by the relay system.

  In practice, a shorter delay $\Delta_{challengeRedeem}$ may be defined within which the vault can challenge the burn transaction, as they may do so as soon as it is included in I, but we choose not to do so for simplicity.

- ICN_w is the warranty collateral. This is a small amount of collateral that issuers must lock for the duration of an Issue procedure in order to prevent griefing attacks, in which an attacker submits multiple requests to lock funds with vaults without following through. Similarly, redeemers must lock this amount when submitting a burn transaction

such as to prevent spam attacks, in which they repeatedly request to redeem but do not properly encrypt the requested note to the vaults.

This amount is also deducted from a vault's collateral if they fail to react in time at any step of the protocols.

- ℓ_f is the logarithm with base 2 of the fee granularity, as defined for f.
- ℓ_{value^I} is defined assuming that the largest possible ICN value can be represented as $ICN_{\max} = 2^{\ell_{\mathsf{value}^I}} - 1$.
- xr_G is the exchange rate granularity. Exchange rates on Zclaim are always represented as $\frac{\mathsf{xr}_n}{\mathsf{xr}_G}$.
- ℓ_{xr_G} is defined such that $\mathsf{xr}_G = 2^{\ell_{\mathsf{xr}_G}}$.
- ℓ_{xr_n} is the bit length of the exchange rate numerator such that $\mathsf{xr}_n : \{0 \,..\, 2^{\ell_{\mathsf{xr}_n}} - 1$. This value should be chosen such that $\frac{2^{\ell_{\mathsf{xr}_n}}-1}{2^{\ell_{\mathsf{xr}_G}}}$ is large enough to represent a maximum exchange rate above which the protocol collapses, as discussed in Section 6.2.7.

5.9 Data structures

5.9.1 Vault representation on I

A vault V is represented in the vault registry on I by the tuple $(\mathsf{d}, \mathsf{pk_d}, \mathsf{coll}, \mathsf{cb}, \mathsf{xr}_n, \mathsf{accepts\_issue}, \mathsf{accepts\_redeem})$, where

- $(\mathsf{d} : \mathbb{B}^{[\ell_\mathsf{d}]}, \mathsf{pk_d} : \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{PublicPrimeSubgroup})$ is V's diversified payment address on Zcash as provided by themselves upon registration.
- $\mathsf{coll} : \{0 \,..\, 2^{\ell_{\mathsf{value}^I}} - 1\}$ is the amount of collateral in ICN that V has locked on I.
- $\mathsf{cb} : \mathsf{ValueCommit.Output}$ is a value commitment to the amount of ZEC obligations associated with V. This value is calculated by aggregating the homomorphic value commitments to the ICZ value of all Mint and Burn transfers in which V was involved, employing the abelian group operation on public keys defined in [17, Section 4.1.6.2] as instantiated for RedJubjub[6].
- $\mathsf{xr}_n : \{0 \,..\, 2^{\ell_{\mathsf{xr}_n}} - 1$ is the last exchange rate nominator used by the vault in a proof of balance. This is used to trigger liquidation when the exchange rate provided by the oracle drops by a certain margin below $\frac{\mathsf{xr}_n}{\mathsf{xr}_G}$.

[6]For Burn transfers, the value $-\mathsf{cv}$ where $-\mathsf{cv} + \mathsf{cv} = \mathcal{O}_+$, the group identity, is used instead of cv.

- accepts_issue : $\mathbb{B}$ represents whether V is available to issue funds, i.e. accepts lock transactions, where $1 := \top$ and $0 := \bot$.
- accepts_redeem : $\mathbb{B}$ represents whether V is available to redeem funds, i.e. can submit release transactions, where $1 := \top$ and $0 := \bot$.

5.9.2 Lock permits

A lock permit is represented as $(\mathsf{emk}, \mathsf{d}, \mathsf{pk_d}, \mathsf{n_{permit}})$, where

- emk : MintingSig.Public is an ephemeral issuing public key provided by the issuer. emk is a Jubjub public key.
- $(\mathsf{d} : \mathbb{B}^{[\ell_\mathsf{d}]}, \mathsf{pk_d} : \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{PublicPrimeSubgroup})$ is the vault's diversified payment address as defined in Section 2.2.2.
- $\mathsf{n_{permit}} : \mathbb{B}^{\mathbb{Y}^{[32]}}$ is a cryptographic nonce. This is the digest of a BLAKE2b-256 hash.

5.9.3 Mint transfers

A Mint transfer is similar to a Spend transfer in that it allows the issuer to spend some value v, but instead of proving ownership of a note of this value on I, the issuer is required to show that they have locked an equivalent amount of funds on Zcash. It consists of $(\mathsf{cv}, \mathsf{cv_n}, \mathsf{cr}, \mathsf{emk}, \mathsf{d}, \mathsf{pk_d}, \mathsf{n_{permit}}, \mathsf{cm}_{u\mathbf{n}}, \mathsf{pos_n}, \mathsf{rt_n}, \mathsf{path_n}, \mathsf{epk_n}, \mathsf{C^{enc}}_\mathbf{n}, \mathtt{mintSig}, \pi_{\mathsf{ZKMint}})$, where:

- cv : ValueCommit.Output is a commitment to the ICZ value v being minted.
- $\mathsf{cv_n}$: ValueCommit.Output is a commitment to the ZEC value $\mathsf{v_n}$ that the issuer has locked with the vault on Zcash.
- cr : ValueCommit.Output is a commitment to the fee rounding error. This value is used to verify that the relation between v and $\mathsf{v_n}$ holds based on the value commitments cv and $\mathsf{cv_n}$.
- emk : MintingSig.Public is the minting public key in the lock permit allowing the issuer to lock funds with this vault.
- $(\mathsf{d} : \mathbb{B}^{[\ell_\mathsf{d}]}, \mathsf{pk_d} : \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{PublicPrimeSubgroup})$ is the vault's shielded payment address in the lock permit.
- $\mathsf{n_{permit}} : \mathbb{B}^{\mathbb{Y}^{[32]}}$ is the nonce in the lock permit.
- $\mathsf{cm}_{u\mathbf{n}} : \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]}$ is the note commitment hash of the note the issuer sent to the vault.
- $\mathsf{pos_n} : \{0 \,..\, 2^{\mathsf{MerkleDepth}^{\mathsf{Sapling}}} - 1\}$ is the position of $\mathsf{cm}_{u\mathbf{n}}$ in the note commitment tree.

- $\mathsf{rt}_{\mathbf{n}} : \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]}$ is an anchor for the output treestate of a Zcash block.
- $\mathsf{path}_{\mathbf{n}} : (\mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]})^{[\mathsf{MerkleDepth}^{\mathsf{Sapling}}]}$ is a Merkle path as defined in [17, Section 4.8] from $\mathsf{cm}_{u\,\mathbf{n}}$ to $\mathsf{rt}_{\mathbf{n}}$.
- $\mathsf{epk}_{\mathbf{n}} : \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Public}$ is the ephemeral key agreement public key used to derive the key for the encryption of the transmitted note ciphertext component $\mathsf{C}^{\mathsf{enc}}{}_{\mathbf{n}}$.
- $\mathsf{C}^{\mathsf{enc}}{}_{\mathbf{n}} : \mathsf{Sym}.\mathbf{C}$ is the encrypted note plaintext $(\mathsf{d}, \mathsf{v}, \mathsf{rcm}, \mathsf{memo})$ of the note with note commitment $\mathsf{cm}_{u\,\mathbf{n}}$ encrypted as defined in [17, Section 4.17.1]. The vault needs the values d, rcm in order to be able to spend the aforementioned note. The memo field is irrelevant to the Zclaim protocol.
- `mintSig` : MintingSig.Signature is a signature over the SIGHASH transaction hash that must be verifiable using emk.
- π_{ZKMint} : ZKMint.Proof is a Groth16 zk-SNARK with primary input $(\mathsf{cv}, \mathsf{cv}_{\mathbf{n}}, \mathsf{cr}, \mathsf{g_d}\star, \mathsf{pk_d}\star, \mathsf{n}_{\mathsf{permit}}, \mathsf{cm}_{u\,\mathbf{n}})$, where $\mathsf{g_d}\star = \mathsf{repr}_{\mathbb{J}}(\mathsf{DiversifyHash}(\mathsf{d}))$ and $\mathsf{pk_d}\star = \mathsf{repr}_{\mathbb{J}}(\mathsf{pk_d})$. π_{ZKMint} is a proof for a Mint statement, defined in Section 5.10.3.

5.9.4 Burn transfers

Similarly, a Burn transfer is similar to an Output transfer. As the name suggests, it allows a redeemer to burn assets and redeem their value on Zcash. It consists of $(\mathsf{cv}, \mathsf{cv}_{\mathbf{n}}, \mathsf{cr}, \mathsf{d}, \mathsf{pk_d}, \mathsf{cm}_{u\,\mathbf{n}}, \mathsf{epk}, \mathsf{N}^{\mathsf{enc}}, \pi_{\mathsf{ZKBurn}})$, where:

- cv : ValueCommit.Output is a commitment to the ICZ value v being burned.
- $\mathsf{cv}_{\mathbf{n}}$: ValueCommit.Output is a commitment to the ZEC value $\mathsf{v}_{\mathbf{n}}$ that the redeemer requests the vault to release.
- cr : ValueCommit.Output is a commitment to the fee rounding error. This value is used to verify that the relation between v and $\mathsf{v}_{\mathbf{n}}$ holds based on the value commitments cv and $\mathsf{cv}_{\mathbf{n}}$.
- $(\mathsf{d} : \mathbb{B}^{[\ell_{\mathsf{d}}]}, \mathsf{pk_d} : \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{PublicPrimeSubgroup})$ represent the vault the redeemer is requesting to release funds.
- $\mathsf{cm}_{u\,\mathbf{n}} : \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]}$ is the note commitment hash of the note the redeemer requests to receive on Zcash.
- $\mathsf{epk} : \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Public}$ is the ephemeral key agreement public key used to derive the key for the encryption of the note with note commitment $\mathsf{cm}_{u\,\mathbf{n}}$ to the vault.

- $\mathsf{N}^{\mathsf{enc}}$: Sym.**C** is encryption of the note values of the note with note commitment $\mathsf{cm}_{u_{\mathbf{n}}}$ as defined in [17, Section 4.17.1], where the note plaintext $\mathbf{np}$ is simply replaced by the note $\mathbf{n}$.
- π_{ZKBurn} : ZKBurn.Proof is a Groth16 zk-SNARK with primary input $(\mathsf{cv}, \mathsf{cv}_{\mathbf{n}}, \mathsf{cr}, \mathsf{cm}_{u_{\mathbf{n}}})$ for a Burn statement, defined in Section 5.11.1.

5.10 Issuing

We now provide a specification for the Issue protocol based on the operations introduced in previous sections. For each operation, we define pre-execution steps, a transaction, requirements for that transaction to succeed, and the outcome in case of success. We define the protocol between two actors: an issuer ISSUER and a vault VAULT.

5.10.1 *requestLock*

Pre-execution steps ISSUER queries the vault registry for a vault available to issue, i.e. for which $\mathsf{accepts\_issue} = \top$. They generate a MintingSig private/public key pair $(\mathsf{emsk} : \mathsf{MintingSig.Private}, \mathsf{emk.Public})$ as follows:

· Let MintingSig be the signature scheme defined in Appendix A.2.1
· Let $\mathsf{emsk} = \mathsf{MintingSig.GenPrivate}()$
· Let $\mathsf{emk} = \mathsf{MintingSig.DerivePublic}(\mathsf{emsk})$

Transaction ISSUER submits a requestLock transaction on I, which locks ICN_w and takes $(\mathsf{emk}, \mathsf{d}, \mathsf{pk_d}, \texttt{requestLockSig})$ as input, where

- emk : MintingSig.Public is the public component of the key pair generated by ISSUER.
- $(\mathsf{d} : \mathbb{B}^{[\ell_\mathsf{d}]}, \mathsf{pk_d} : \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{PublicPrimeSubgroup})$ is the diversified payment address of the vault VAULT selected by the issuer.
- `requestLockSig` : MintingSig.Signature is a signature over this transaction, computed as follows:
  · Let RequestLockHash be the digest of an unspecified hash function on the canonical encodings of all other fields in this transaction.
  · Then, $\texttt{requestLockSig} = \mathsf{MintingSig.Sign}_{\mathsf{emsk}}(\mathsf{RequestLockHash})$

Requirements A requestLock transaction is successful if:

- ISSUER's ICN balance is larger than ICN_w.
- $\mathsf{accepts\_issue} = \top$ for vault VAULT with diversified payment address $(\mathsf{d}, \mathsf{pk_d})$.

- emk has not been used in a previous requestLock transaction in this block.
- $\mathsf{MintingSig.Validate}_{\mathsf{emk}}(\mathsf{RequestLockHash}, \texttt{requestLockSig}) = 1$.

Outcome accepts_issue is set to $\perp$ for VAULT and a lock permit is issued containing $(\mathsf{emk}, \mathsf{d}, \mathsf{pk_d}, \mathsf{n_{permit}})$, where

- $\mathsf{emk}, \mathsf{d}, \mathsf{pk_d}$ are the values submitted by ISSUER in the requestLock transaction.
- $\mathsf{n_{permit}}$ is a cryptographic nonce that ISSUER must use to derive the note commitment trapdoor in the note they send to VAULT. This value is generated as follows:
  - Let PREV_HEADER be the concatenation of the encoding of all fields in the previous block header on I, i.e. the header of the block immediately preceding the one in which the lock permit is to be included.
  - Then, $\mathsf{n_{permit}} = \mathsf{BLAKE2b\text{-}256}(\text{“}\texttt{ZCLAIM\_lock\_perm}\text{”}, \mathsf{PREV\_HEADER} \,||\, \mathsf{emk})$

  Since it is verified that no two ephemeral issuing keys in lock requests in the same block are the same, this value is unique.

If ISSUER does not submit a mint transaction within Δ_{mint}, the lock permit is discarded and ISSUER's warranty collateral ICN_w is transferred to VAULT.

5.10.2 *lock*

Pre-execution steps ISSUER waits until the preceding requestLock transaction is securely included on I. They craft a note $\mathbf{n}$ to send to VAULT with destination diversified payment address $(\mathsf{d}, \mathsf{pk_d})$, value $\mathsf{v_n} \leq \mathsf{v_{max}}$ and note commitment trapdoor $\mathsf{rcm_n}$ generated as follows:

- Let NonceCommit be the nonce commitment scheme defined in Appendix A.1.1
- Choose a uniformly random nonce commitment trapdoor $\mathsf{rcn} \xleftarrow{R} \mathsf{NonceCommit.GenTrapdoor}()$
- Then, $\mathsf{rcm_n} = \mathsf{NonceCommit}_{\mathsf{rcn}}(\mathsf{n_{permit}})$

Transaction The lock transaction that ISSUER must submit on Zcash may be any transaction in which $\mathbf{n}$ is the output note of an Output transfer. We call this Output transfer the lock transfer. Besides, the lock transaction may contain any number of transparent inputs and outputs, Spend transfers and other Output transfers.

Requirements The lock transaction succeeds if it follows the standard Zcash protocol rules. See the specification for more details.

Outcome This transaction has no immediate effect on I.

5.10.3 *mint*

Pre-execution steps ISSUER waits until the lock transaction has been securely included on Zcash and confirmed by the relay system.

They calculate the ICZ value to be minted after deduction of fees as $\mathsf{v} = \lfloor \mathsf{v_n} \cdot (1 - f) \rfloor$ and construct a Mint transfer as defined in Section 5.9.3 using the following values:

- cv must be a value commitment to the ICZ value being minted, in which the randomness must be derived from the note commitment trapdoor of note $\mathbf{n}$ as follows:
  - Let $\mathsf{rcv} = \mathsf{LEOS2IP}_{256}(\mathsf{BLAKE2b\text{-}256}(\text{“}\texttt{ZCLAIM\_derive\_cv}\text{”}, \mathsf{rcm_n})) \mod 2^{251}$
  - Then, $\mathsf{cv} = \mathsf{ValueCommit}_{\mathsf{rcv}}(\mathsf{v})$
- $\mathsf{cv_n}$ is a value commitment for $\mathsf{v_n}$, the ZEC value locked with VAULT. Generally, this will be the value commitment cv in the lock transfer. Otherwise, it is to be generated in the same way as the aforementioned as described in [17, Section 4.6.2], i.e.:
  - Choose a uniformly random commitment trapdoor $\mathsf{rcv_n} \xleftarrow{R} \mathsf{ValueCommit.GenTrapdoor}()$
  - Then, $\mathsf{cv_n} = \mathsf{ValueCommit}_{\mathsf{rcv_n}}(\mathsf{v_n})$
- cr is a value commitment to the fee rounding error. This value must be computed as $\mathsf{cr} = [f_G]\,\mathsf{cv} - [f_G - f_n]\,\mathsf{cv_n}$.
- $\mathsf{emk}, \mathsf{d}, \mathsf{pk_d}, \mathsf{n_{permit}}$ are the values in the lock transfer.
- $\mathsf{cm}_{u\mathbf{n}}$ must be the note commitment hash of note $\mathbf{n}$, generated as described in [17, Section 4.6.2] as follows:
  - Let $\mathsf{g_d} = \mathsf{DiversifyHash}(\mathsf{d})$
  - Then, $\mathsf{cm}_{u\mathbf{n}} = \mathsf{Extract}_{\mathbb{J}^{(r)}}(\mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm_n}}(\mathsf{repr}_{\mathbb{J}}(\mathsf{g_d}), \mathsf{repr}_{\mathbb{J}}(\mathsf{pk_d}), \mathsf{v_n}))$

  This value is published in the lock transfer, at which point it is added to Zcash's note commitment tree.
- $\mathsf{pos_n}$ is the index of the leaf node occupied by the hash value $\mathsf{cm}_{u\mathbf{n}}$ in Zcash's note commitment tree. This value is obtained by examining the note commitment tree after the lock transaction has been included in a block.
- $\mathsf{rt_n}$ must be an anchor for the output treestate of a Zcash block of block height $h \geq h_{\mathbf{n}}$, where $h_{\mathbf{n}}$ is the height of the block in which $\mathbf{n}$ was created.

- $\mathsf{path}_{\mathbf{n}}$ is a Merkle path from the leaf node containing $\mathsf{cm}_{u\,\mathbf{n}}$ to $\mathsf{rt}_{\mathbf{n}}$, generated as described in [17, Section 4.8].

- $\mathsf{epk}_{\mathbf{n}}$ is used in the symmetric encryption scheme used to encrypt and decrypt ${\mathsf{C}^{\mathsf{enc}}}_{\mathbf{n}}$.

  Generally, this will be the value epk used in the lock transfer. Otherwise, it is to be generated in the same way as epk as described in [17, Sections 4.6.2 and 4.17.1], i.e.:

  · Choose a uniformly random ephemeral private key $\mathsf{esk}_{\mathbf{n}} \xleftarrow{R} \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Private} \setminus \{0\}$
  · Then $\mathsf{epk}_{\mathbf{n}} = \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{DerivePublic}(\mathsf{esk}_{\mathbf{n}}, \mathsf{DiversifyHash}(\mathsf{d}))$, where d is vault's diversifier

- ${\mathsf{C}^{\mathsf{enc}}}_{\mathbf{n}}$ is the note plaintext of note $\mathbf{n}$ encrypted as defined in [17, Section 4.17.1].

  Generally, this will be the value $\mathsf{C}^{\mathsf{enc}}$ in the lock transfer. Otherwise, it is to be generated in the same way as $\mathsf{C}^{\mathsf{enc}}$ as described in [17, Section 4.17.1] with $\mathsf{esk} = \mathsf{esk}_{\mathbf{n}}$, the private key from which $\mathsf{epk}_{\mathbf{n}}$ was derived.

- `mintSig` is generated as follows:

  · Let SigHash be the SIGHASH transaction hash as defined in Appendix A.3
  · Then, $\texttt{mintSig} = \mathsf{MintingSig.Sign}_{\mathsf{emsk}}(\mathsf{SigHash})$, where emsk is the private key from which emk was derived

- π_{ZKMint} is a zero-knowledge proof for a Mint statement, which proves that, given a primary input

$$
\begin{aligned}
(\mathsf{cv} \quad &: \mathsf{ValueCommit.Output},\\
\mathsf{cv}_{\mathbf{n}} \quad &: \mathsf{ValueCommit.Output},\\
\mathsf{cr} \quad &: \mathsf{ValueCommit.Output},\\
\mathsf{g_d}\star \quad &: \mathbb{B}^{[\ell_{\mathbb{J}}]},\\
\mathsf{pk_d}\star \quad &: \mathbb{B}^{[\ell_{\mathbb{J}}]},\\
\mathsf{n}_{\mathsf{permit}} \quad &: \mathbb{B}^{\mathbb{Y}^{[32]}},\\
\mathsf{cm}_{u\,\mathbf{n}} \quad &: \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]}),
\end{aligned}
$$

the prover knows an auxiliary input

$$\begin{aligned}(&\mathsf{v} &&: \{0 .. \mathsf{v}_{\mathsf{max}}\},\\ &\mathsf{v}_{\mathbf{n}} &&: \{0 .. \mathsf{v}_{\mathsf{max}}\},\\ &\mathsf{rcv}_{\mathbf{n}} &&: \{0 .. 2^{\ell_{\mathsf{scalar}}} - 1\},\\ &\mathsf{rcm}_{\mathbf{n}} &&: \{0 .. 2^{\ell_{\mathsf{scalar}}} - 1\},\\ &\mathsf{rcn}_{\mathbf{n}} &&: \{0 .. 2^{\ell_{\mathsf{scalar}}} - 1\},\\ &\mathsf{rcv} &&: \{0 .. 2^{\ell_{\mathsf{scalar}}} - 1\},\\ &\mathsf{rcr} &&: \{0 .. 2^{\ell_{\mathsf{scalar}}} - 1\},\\ &\mathsf{rem} &&: \{0 .. f_G - 1\})\end{aligned}$$

such that the following conditions hold:

Note commitment integrity
$\mathsf{cm}_{u\,\mathbf{n}} = \mathsf{Extract}_{\mathbb{J}^{(r)}}(\mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}_{\mathbf{n}}}(\mathsf{g_d}\star, \mathsf{pk_d}\star, \mathsf{v}_{\mathbf{n}}))$

Locked value commitment integrity $\mathsf{cv}_{\mathbf{n}} = \mathsf{ValueCommit}_{\mathsf{rcv}_{\mathbf{n}}}(\mathsf{v}_{\mathbf{n}})$

Minted value commitment integrity $\mathsf{cv} = \mathsf{ValueCommit}_{\mathsf{rcv}}(\mathsf{v})$

Fee rounding commitment integrity $\mathsf{cr} = \mathsf{ValueCommit}_{\mathsf{rcr}}(\mathsf{rem})$

Trapdoor commitment integrity $\mathsf{rcv}_{\mathbf{n}} = \mathsf{NonceCommit}_{\mathsf{rcn}_{\mathbf{n}}}(\mathsf{n}_{\mathsf{permit}})$

Apart from these constraints, all auxiliary inputs must be range checked and some small order checks may be necessary.

The purpose of the **Trapdoor commitment integrity** condition is to ensure that note **n** was created expressly for the purpose of this Issue procedure. Alternatively, a collection containing positioned Zcash notes used in past Issue procedures could be implemented on I, which would however lead to a larger strain on storage and computational power due to the insertion and searching of notes in this data structure.

The concrete implementation of this zk-SNARK falls beyond the scope of this work, but the above conditions have been chosen such that they can be implemented using circuit components already in use in the Sapling zk-SNARKs, as per the stated **Minimal modifications** design goal.

ISSUER instantiates this zk-SNARK with auxiliary inputs $\mathsf{rcr} = f_G \cdot \mathsf{rcv} - (f_G - f_n) \cdot \mathsf{rcv}_{\mathbf{n}}$ and $\mathsf{rem} = f_G \cdot \mathsf{v} - (f_G - f_n) \cdot \mathsf{v}_{\mathbf{n}}$, primary inputs $\mathsf{g_d}\star = \mathsf{repr}_{\mathbb{J}}(\mathsf{DiversifyHash}(\mathsf{d}))$ and $\mathsf{pk_d}\star = \mathsf{repr}_{\mathbb{J}}(\mathsf{pk_d})$ and all other values as previously defined.

Transaction A mint transaction is a transaction on I that contains a Mint transfer and any number of transparent inputs and outputs, and Spend and Output transfers. Ideally, however, it should only contain Spend and Output transfers apart from the Mint transfer, such as not to compromise privacy. See discussion in Section 6.3 on the matter. Usually, it will consist of only the Mint transfer and one Output transfer.

ISSUER submits a mint transaction on I containing the Mint transfer described above.

Requirements The mint transaction succeeds only if the following conditions hold for the Mint transfer:

- $\mathsf{path}_{\mathbf{n}}$ is a Merkle path from $\mathsf{cm}_{u\,\mathbf{n}}$ to $\mathsf{rt}_{\mathbf{n}}$. This is verified as follows:
  - Let $\mathsf{HashValue}_{\mathsf{MerkleDepth}^{\mathsf{Sapling}}} = \mathsf{cm}_{u\,\mathbf{n}}$
  - Let $\mathsf{path}[i]$ be the i-th element of the Merkle path, where $0 \leq i \leq \mathsf{MerkleDepth}^{\mathsf{Sapling}} - 1$
  - For i from $\mathsf{MerkleDepth}^{\mathsf{Sapling}}$ down to 1:
    - Let $\mathsf{parity} = \left\lfloor \frac{\mathsf{pos}_{\mathbf{n}}}{2^{\mathsf{MerkleDepth}^{\mathsf{Sapling}} - i}} \right\rfloor \,\&\, 1$
    - Let $\mathsf{hash} = \mathsf{HashValue}_i$
    - Let $\mathsf{sibling} = \mathsf{path}[\mathsf{MerkleDepth}^{\mathsf{Sapling}} - i]$
    - If $\mathsf{parity} = 0$:
      - Let $\mathsf{HashValue}_{i-1} = \mathsf{MerkleCRH}^{\mathsf{Sapling}}(i - 1, \mathsf{hash}, \mathsf{sibling})$
    - Else:
      - Let $\mathsf{HashValue}_{i-1} = \mathsf{MerkleCRH}^{\mathsf{Sapling}}(i - 1, \mathsf{sibling}, \mathsf{hash})$
  - Then, return $\top$ if $\mathsf{HashValue}_0 = \mathsf{rt}_{\mathbf{n}}$ and $\bot$ otherwise
- $\mathsf{rt}_{\mathbf{n}}$ is an anchor for the output treestate of a Zcash block that has been confirmed by the relay system. In Sapling, $\mathsf{rt}_{\mathbf{n}}$ is encoded in block headers as the field `hashFinalSaplingRoot`.

  As a side note, this field was replaced with a commitment to a larger data structure that in turn commits to the note commitment tree [51] in the recent Heartwood upgrade [58] mentioned in Section 5.4.1. This data structure is a Merkle Mountain Range (MMR) that also commits to several other features of the chain's history. Hence, showing that $\mathsf{rt}_{\mathbf{n}}$ is indeed an anchor for the output treestate of a post-Heartwood block would imply providing a path from $\mathsf{rt}_{\mathbf{n}}$ to the root of the MMR.
- The values $\mathsf{emk}, \mathsf{d}, \mathsf{pk}_{\mathsf{d}}, \mathsf{n}_{\mathsf{permit}}$ match those in a lock permit.
- The relation

$$\mathsf{cr} = [f_G]\,\mathsf{cv} - [f_G - f_n]\,\mathsf{cv}_{\mathbf{n}} \tag{5.2}$$

  holds.

As per the specification of ValueCommit in [17, Section 5.4.7.3], this expands to

$$\begin{aligned}\mathsf{cr} &= [f_G]\,([\mathsf{v}]\,\mathcal{V} + [\mathsf{rcv}]\,\mathcal{R}) - [f_G - f_n]\,([\mathsf{v_n}]\,\mathcal{V} + [\mathsf{rcv_n}]\,\mathcal{R}) \\ &= [f_G \cdot \mathsf{v} - (f_G - f_n) \cdot \mathsf{v_n}]\,\mathcal{V} + [f_G \cdot \mathsf{rcv} - (f_G - f_n) \cdot \mathsf{rcv_n}]\,\mathcal{R} \\ &= [\mathsf{rem}']\,\mathcal{V} + [\mathsf{rcr}']\,\mathcal{R} \\ &= \mathsf{ValueCommit}_{\mathsf{rcr}'}(\mathsf{rem}')\end{aligned}$$

Since π_{ZKMint} proves that $\mathsf{cr} = \mathsf{ValueCommit}_{\mathsf{rcr}}(\mathsf{rem})$ for some $\mathsf{rem} < f_G$, it follows from Eq. (5.2) and the security requirements of ValueCommit, which must be computationally binding, that

$$\begin{aligned} f_G \cdot \mathsf{v} - (f_G - f_n) \cdot \mathsf{v_n} = \mathsf{rem}' = \mathsf{rem} \leq f_G \\ \Leftrightarrow \quad \mathsf{v} = \lfloor \mathsf{v_n} \cdot (1 - f) \rfloor \end{aligned}$$

Hence this check ensures that the minted value was calculated correctly. The verification is done outside of π_{ZKMint} for efficiency reasons.

- π_{ZKMint} is verified to be correct as described in [17, Appendix B.2].
- $\mathsf{MintingSig.Validate}_{\mathsf{emk}}(\mathsf{SigHash}, \texttt{mintSig}) = 1$.

Outcome This operation creates a mint transaction that is published but not included in the block chain until either VAULT executes *confirmIssue* or the delay $\Delta_{confirmIssue}$ has passed. If VAULT executes *challengeIssue* within said delay, the transaction is discarded.

If no confirmIssue or challengeIssue transaction is submitted within this delay, the outcome is still the same as that of a confirmIssue operation, but additionally, ICN_w is deducted from VAULT's collateral and transferred to ISSUER. In that case, we assume that ISSUER has no way of knowing in advance that VAULT will not be able to react and thus must have encrypted the note plaintext correctly.

The corresponding lock permit is immediately discarded.

5.10.4 *confirmIssue*

Pre-execution steps VAULT sees the pending mint transaction on I containing a Mint transfer that includes their diversified payment address $(\mathsf{d}, \mathsf{pk_d})$. They proceed to verify the validity of said Mint transfer as follows:

1. **Note ciphertext verification.**

   If VAULT has received the note plaintext of note $\mathbf{n}$ with note commitment $\mathsf{cm}_{u\mathbf{n}}$ in the lock transaction, they can skip this step.

Otherwise, they decrypt $C^{enc}_{\mathbf{n}}$ as described in [17, Section 4.17.2] with parameters `ephemeralKey`, C^{enc} and `cmu` being the encodings of $epk_{\mathbf{n}}$, $C^{enc}_{\mathbf{n}}$ and $cm_{u\mathbf{n}}$, respectively. There is no need to verify that VAULT is the recipient of the note as this is already proven in the zk-SNARK.

2. **Value commitment randomness verification.**

   VAULT needs to know the trapdoor used to generate the value commitment cv in the Mint transfer in order to be able to submit proofs of balance including said transfer.

   Hence they verify that the trapdoor was derived from **n**'s note commitment trapdoor as follows:

   · Let $rcm_{\mathbf{n}}$ and $v_{\mathbf{n}}$ be the note commitment trapdoor and value in the decrypted note plaintext $\mathbf{np}$
   · Let $v = \lfloor v_{\mathbf{n}} \cdot (1 - f) \rfloor$ be the minted value
   · Let $rcv' = \mathsf{LEOS2IP}_{256}(\mathsf{BLAKE2b\text{-}256}(\text{“}\texttt{ZCLAIM\_derive\_cv}\text{”}, rcm_{\mathbf{n}})) \mod 2^{251}$
   · Let $cv' = \mathsf{ValueCommit}_{rcv'}(v)$
   · Then, return $\top$ if $cv = cv'$, otherwise return $\bot$

If any of these steps fails, VAULT should execute *challengeIssue* instead.

Transaction VAULT submits a confirmIssue transaction on I, which takes (d, pk_d, n_{permit}, `issueConfSig`) as input, where:

- d, pk_d, n_{permit} must be the values in the Mint transfer to be confirmed.
- `issueConfSig` : VaultSig.Signature is a signature over the hashes of this transaction and the Mint transfer, computed as follows:

  · Let MintHash be a hash of the canonical encoding of all fields in the Mint transfer
  · Let ConfirmIssueHash be a hash of the canonical encoding of all other fields in the confirmIssue transaction
  · Let ivk be VAULT's incoming viewing key
  · Let VaultSig be the signature scheme defined in Appendix A.2.2
  · Then, $\texttt{issueConfSig} = \mathsf{VaultSig.Sign}_{ivk}(\mathsf{ConfirmIssueHash} \,||\, \mathsf{MintHash})$

  The input to this signature includes the hash of the Mint transfer since the confirmIssue transaction may otherwise be replayed over a different Mint transfer in which the note values have not been correctly encrypted to VAULT in a double spend attack. If VAULT has not saved these values when confirming the original transfer, they will not have access to the locked funds.

Requirements The confirmIssue transaction succeeds if:

- The values $\mathsf{d}, \mathsf{pk_d}, \mathsf{n_{permit}}$ match those in a pending Mint transfer.
- $\mathsf{VaultSig.Validate_{pk_d}}(\mathsf{ConfirmIssueHash} \mid\mid \mathsf{MintHash}, \texttt{issueConfSig}) = 1$.

Outcome The mint transaction is confirmed and the minted funds become immediately available to ISSUER.

The ZEC obligations value commitment cb associated with VAULT is updated as follows:

· Let cv be the ICZ value commitment in the Mint transfer
· Let '+' be the abelian group operation on public keys defined in [17, Section 4.1.6.2] as instantiated for RedJubjub
· Then, let $\mathsf{cb}^{\mathsf{new}} = \mathsf{cb}^{\mathsf{old}} + \mathsf{cv}$

The warranty collateral ICN_w is returned to ISSUER.

5.10.5 *challengeIssue*

Pre-execution steps VAULT has established the invalidity of the pending Mint transfer as per the pre-transaction steps of *confirmIssue*.

Transaction A challengeIssue transaction on I takes $(\mathsf{sharedSecret}, \mathsf{d}, \mathsf{pk_d}, \mathsf{epk_n}, \pi_{\mathsf{ZKChallenge}})$ as input, where:

- $\mathsf{sharedSecret} : \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{SharedSecret}$ is the shared secret presumably used by ISSUER to encrypt $\mathsf{C}^{\mathsf{enc}}_{\mathbf{n}}$. This value is computed by VAULT as $\mathsf{sharedSecret} = \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Agree}(\mathsf{ivk}, \mathsf{epk_n})$.
- d, $\mathsf{pk_d}$, $\mathsf{epk_n}$ must be the values in the Mint transfer.
- $\pi_{\mathsf{ZKChallenge}} : \pi_{\mathsf{ZKChallenge}}.\mathsf{Proof}$ is a Groth16 zk-SNARK for a Challenge statement, which proves that, given a primary input

$$\begin{aligned}(&\mathsf{sharedSecret} : \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{SharedSecret},\\ &\mathsf{g_d} \quad\quad\quad : \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{PublicPrimeSubgroup},\\ &\mathsf{pk_d} \quad\quad\quad : \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{PublicPrimeSubgroup},\\ &\mathsf{epk} \quad\quad\quad : \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Public})\end{aligned}$$

  the prover knows an auxiliary input

$$(\mathsf{ivk} \quad\quad\quad : \{0 .. 2^{\ell_{\mathsf{ivk}}} - 1\})$$

  such that the following conditions hold:

  Diversified address integrity $\mathsf{pk_d} = [\mathsf{ivk}]\,\mathsf{g_d}$

  Shared secret integrity $\mathsf{sharedSecret} = [h_{\mathbb{J}} \cdot \mathsf{ivk}]\,\mathsf{epk}$

Effectively, this statement is a non-interactive proof of discrete-logarithm equality or DLEQ, with input pairs $(\mathsf{pk_d}, \mathsf{g_d})$ and $(\mathsf{sharedSecret}, [h_{\mathbb{J}}]\,\mathsf{epk_n})$. This proof allows VAULT to show that the revealed shared secret was correctly generated using their incoming viewing key without revealing the latter.

VAULT must instantiate this zk-SNARK with primary inputs $(\mathsf{sharedSecret}, \mathsf{DiversifyHash(d)}, \mathsf{pk_d}, \mathsf{epk_n})$.

This transaction does not contain a signature since the vault already proves knowledge of their own incoming viewing key ivk in $\pi_{\mathsf{ZKChallenge}}$. Furthermore, it is not necessary to commit to the Mint transfer since a challengeIssue transaction may only be replayed over a different Mint transfer if the encryption of $\mathsf{C^{enc}_n}$ is also erroneous in said transfer, which does not represent undesirable behaviour.

Requirements The challengeIssue transaction succeeds if:

- The values d, $\mathsf{pk_d}$, $\mathsf{epk_n}$ match those in a pending Mint transfer.

  Note that the values $(\mathsf{d}, \mathsf{pk_d})$ are enough to uniquely identify the Mint transfer, since ZCLAIM currently only allows for one concurrent Issue procedure per vault. If the protocol is expanded to allow multiple concurrent instances of Issue per vault, the value $\mathsf{n_{permit}}$ should be added to the challengeRedeem transaction inputs as there is no mechanism in place to prevent $\mathsf{epk_n}$ from begin reused across multiple Issue transfers.

- The verification of $\pi_{\mathsf{ZKChallenge}}$ as described in [17, Appendix B.2] succeeds.

- Any of **Note ciphertext verification** or **Value commitment randomness verification** as defined in the pre-execution steps of *confirmIssue* fail.

  This can be verified using the revealed shared secret as follows:

  · Attempt decryption of $\mathsf{C^{enc}_n}$ as described in **Note ciphertext verification**. Skip step 3 in [17, Section 4.17.2] referenced therein and let sharedSecret be the value revealed by VAULT in this transaction.
  · If the decryption algorithm returns $\perp$, return $\top$.
  · Verify that the trapdoor for cv was correctly derived as described in **Value commitment randomness verification**.
  · If the verification algorithm returns $\perp$, return $\top$. Otherwise return $\perp$.

Outcome The challenged mint transaction is discarded and ISSUER's warranty collateral ICN_w is paid to VAULT to compensate for the opportunity

cost.

Note that if the note values were correctly transmitted but VAULT successfully challenged the mint transaction due to an erroneously constructed cv, VAULT retains control over the locked funds. In order to prevent this situation, issuers should adhere to the protocol.

5.11 Redeeming

Similarly, we define the Redeem protocol between a redeemer REDEEMER and a vault VAULT.

5.11.1 *burn*

Pre-execution steps REDEEMER queries the vault registry for a vault available to redeem, i.e. for which $\mathsf{accepts\_redeem} = \top$. They decide on VAULT with diversified payment address $(\mathsf{d}, \mathsf{pk_d})$.

REDEEMER determines an amount $\mathsf{v} \leq \mathsf{v_{max}}$ of ICZ to burn and calculates the amount of ZEC that they can request VAULT to release after deduction of fees as $\mathsf{v_n} = \lfloor \mathsf{v} \cdot (1 - f) \rfloor$. They construct a Zcash note $\mathbf{n}$, which VAULT will be requested to create, with a destination diversified payment address $(\mathsf{d_n}, \mathsf{pk_{d_n}})$ of their choice, value $\mathsf{v_n}$ and randomly generated randomness $\mathsf{rcm_n}$.

Then, they construct a Burn transfer as defined in Section 5.9.4 using the following values:

- cv must be a value commitment to the ICZ value being burned, in which the randomness must be derived from the note commitment trapdoor of note $\mathbf{n}$ in the same way as cv in Mint transfers.
- $\mathsf{cv_n}$ must be a value commitment to the ZEC value to be released $\mathsf{v_n}$, generated as follows:
  - Choose a uniformly random commitment trapdoor $\mathsf{rcv_n} \xleftarrow{R} \mathsf{ValueCommit.GenTrapdoor}()$
  - Then, $\mathsf{cv_n} = \mathsf{ValueCommit}_{\mathsf{rcv_n}}(\mathsf{v_n})$
- cr must be a value commitment to the fee rounding error. This value is computed as $\mathsf{cr} = [f_G]\,\mathsf{cv_n} - [f_G - f_n]\,\mathsf{cv}$.
- $(\mathsf{d}, \mathsf{pk_d})$ is VAULT's diversified payment address.
- $\mathsf{cm}_{u\,\mathbf{n}}$ is generated as follows:
  - Let $\mathsf{g_{d_n}} = \mathsf{DiversifyHash}(\mathsf{d_n})$
  - Then, $\mathsf{cm}_{u\,\mathbf{n}} = \mathsf{Extract}_{\mathbb{J}^{(r)}}(\mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm_n}}(\mathsf{repr}_{\mathbb{J}}(\mathsf{g_{d_n}}), \mathsf{repr}_{\mathbb{J}}(\mathsf{pk_{d_n}}), \mathsf{v_n}))$
- epk is generated as described in [17, Sections 4.6.2 and 4.17.1], i.e.:

  - Choose a uniformly random ephemeral private key $\mathsf{esk} \xleftarrow{R} \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Private} \setminus \{0\}$
  - Then $\mathsf{epk} = \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{DerivePublic}(\mathsf{esk}, \mathsf{DiversifyHash}(\mathsf{d}))$, where d is vault's diversifier

- $\mathsf{N}^{\mathsf{enc}}$ is obtained from $\mathbf{n}$ and v similarly to $\mathsf{C}^{\mathsf{enc}}$ from $\mathbf{np}$ in Output transfers:
  - Let N be the canonical encoding of all fields of note $\mathbf{n}$, i.e. $(\mathsf{d}_{\mathbf{n}}, \mathsf{pk}_{\mathsf{d}\mathbf{n}}, \mathsf{v}_{\mathbf{n}}, \mathsf{rcm}_{\mathbf{n}})$
  - Let $\mathtt{ephemeralKey} = \mathsf{LEBS2OSP}_{\ell_{\mathbb{J}}}(\mathsf{repr}_{\mathbb{J}}(\mathsf{epk}))$
  - Let $\mathsf{sharedSecret} = \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Agree}(\mathsf{esk}, \mathsf{pk}_{\mathsf{d}})$
  - Let $\mathsf{K}^{\mathsf{enc}} = \mathsf{KDF}^{\mathsf{Sapling}}(\mathsf{sharedSecret}, \mathtt{ephemeralKey})$
  - Then, $\mathsf{N}^{\mathsf{enc}} = \mathsf{Sym.Encrypt}_{\mathsf{K}^{\mathsf{enc}}}(\mathsf{v} \,||\, \mathsf{N})$

- π_{ZKBurn} is a zero-knowledge proof for a Burn statement, which proves that, given a primary input

$$\begin{aligned}(&\mathsf{cv} && : \mathsf{ValueCommit.Output},\\ &\mathsf{cv}_{\mathbf{n}} && : \mathsf{ValueCommit.Output},\\ &\mathsf{cr} && : \mathsf{ValueCommit.Output},\\ &\mathsf{cm}_{u\mathbf{n}} && : \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]}),\end{aligned}$$

  the prover knows an auxiliary input

$$\begin{aligned}(&\mathsf{v} && : \{0 \,..\, \mathsf{v}_{\mathsf{max}}\},\\ &\mathsf{v}_{\mathbf{n}} && : \{0 \,..\, \mathsf{v}_{\mathsf{max}}\},\\ &\mathsf{rcv} && : \{0 \,..\, 2^{\ell_{\mathsf{scalar}}} - 1\},\\ &\mathsf{rcv}_{\mathbf{n}} && : \{0 \,..\, 2^{\ell_{\mathsf{scalar}}} - 1\},\\ &\mathsf{g}_{\mathsf{d}}\star_{\mathbf{n}} && : \mathbb{B}^{[\ell_{\mathbb{J}}]},\\ &\mathsf{pk}_{\mathsf{d}}\star_{\mathbf{n}} && : \mathbb{B}^{[\ell_{\mathbb{J}}]},\\ &\mathsf{rcm}_{\mathbf{n}} && : \{0 \,..\, 2^{\ell_{\mathsf{scalar}}} - 1\},\\ &\mathsf{rem} && : \{0 \,..\, f_G - 1\},\\ &\mathsf{rcr} && : \{0 \,..\, 2^{\ell_{\mathsf{scalar}}} - 1\})\end{aligned}$$

  such that the following conditions hold:

  Burned value commitment integrity $\mathsf{cv} = \mathsf{ValueCommit}_{\mathsf{rcv}}(\mathsf{v})$

  Requested value commitment integrity $\mathsf{cv}_{\mathbf{n}} = \mathsf{ValueCommit}_{\mathsf{rcv}_{\mathbf{n}}}(\mathsf{v}_{\mathbf{n}})$

  Requested note commitment integrity
  $\mathsf{cm}_{u\mathbf{n}} = \mathsf{Extract}_{\mathbb{J}^{(r)}}(\mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}_{\mathbf{n}}}(\mathsf{g}_{\mathsf{d}}\star_{\mathbf{n}}, \mathsf{pk}_{\mathsf{d}}\star_{\mathbf{n}}, \mathsf{v}_{\mathbf{n}}))$

  Fee rounding commitment integrity $\mathsf{cr} = \mathsf{ValueCommit}_{\mathsf{rcr}}(\mathsf{rem})$

REDEEMER computes the auxiliary inputs rcr and rem as $\mathsf{rcr} = f_G \cdot \mathsf{rcv}_{\mathbf{n}} - (f_G - f_n) \cdot \mathsf{rcv}$ and $\mathsf{rem} = f_G \cdot \mathsf{v}_{\mathbf{n}} - (f_G - f_n) \cdot \mathsf{v}$.

This zk-SNARK can also be implemented using circuit components already in use in the Sapling zk-SNARKs.

A burn transaction also requires REDEEMER to lock ICN_w.

Burn transfers do not contain a signature themselves as they are signed among all other inputs and outputs in the `spendAuthSig` and `mintSig` signatures of Spend and Mint transfers, respectively.

Transaction REDEEMER submits a burn transaction on I containing a Burn transfer constructed as described above.

A burn transaction, like mint transactions, may contain any number of other Sapling inputs and outputs. Usually, it will consist of a number of Spend transfers and the Burn transfer.

Requirements The burn transaction succeeds only if:

- REDEEMER's ICN balance is larger than ICN_w.
- $\mathsf{accepts\_redeem} = \top$ for vault VAULT with diversified payment address $(\mathsf{d}, \mathsf{pk_d})$.
- The following conditions hold for the Burn transfer:
  - The relation $\mathsf{cr} = [f_G]\,\mathsf{cv}_{\mathbf{n}} - [f_G - f_n]\,\mathsf{cv}$ holds.

    As argued in the transaction requirements of *mint*, this proves that the requested value $\mathsf{v}_{\mathbf{n}}$ was calculated correctly.
  - π_{ZKBurn} is verified to be correct as described in [17, Appendix B.2].

Outcome This operation creates a burn transaction that is published but not included in the block chain unless VAULT executes *confirmRedeem* within $\Delta_{confirmRedeem}$. If VAULT executes *challengeRedeem* instead, the transaction is discarded. Lastly, if VAULT fails to react within $\Delta_{confirmRedeem}$:

- The burn transaction is discarded.
- The warranty collateral ICN_w locked in this transaction is returned to REDEEMER.
- ICN_w is deducted from VAULT's collateral and transferred to REDEEMER.
- If $\mathsf{accepts\_issue} = \top$ for VAULT, it is set to $\bot$. This is because their collateral has been decreased and thus they need to provide a new proof of balance.

$\mathsf{accepts\_redeem}$ is set to $\bot$ for VAULT.

5.11.2 *release*

Pre-execution steps vault sees the pending burn transaction on I containing a Burn transfer that includes their diversified payment address $(\mathsf{d}, \mathsf{pk_d})$. They proceed to verify the validity of said Burn transfer as follows:

1. **Note commitment verification.**

   vault decrypts $\mathsf{N^{enc}}$ similarly to $\mathsf{C^{enc}}$ in incoming Output transfers and then verifies that the extracted note has note commitment $\mathsf{cm}_{u\,\mathbf{n}}$:

   · Let $\mathsf{sharedSecret} = \mathsf{KA^{Sapling}.Agree}(\mathsf{ivk}, \mathsf{epk})$, where ivk is vault's incoming viewing key
   · Let $\mathsf{K^{enc}} = \mathsf{KDF^{Sapling}}(\mathsf{sharedSecret}, \mathsf{LEBS2OSP}_{\ell_{\mathbb{J}}}(\mathsf{repr}_{\mathbb{J}}(\mathsf{epk})))$
   · Let $\mathsf{v}' \,||\, \mathsf{N} = \mathsf{Sym.Decrypt}_{\mathsf{K^{enc}}}(\mathsf{N^{enc}})$
   · Extract $\mathsf{v}' : \{0 \,..\, \mathsf{v_{max}}\}$ and $\mathbf{n}' = (\mathsf{d}'_{\mathbf{n}} : \mathbb{B}^{[\ell_{\mathsf{d}}]}, \mathsf{pk}'_{\mathbf{d}\,\mathbf{n}} : \mathsf{KA^{Sapling}.PublicPrimeSubgroup}, \mathsf{v}'_{\mathbf{n}} : \{0 \,..\, \mathsf{v_{max}}\}, \mathsf{rcm}'_{\mathbf{n}} : \{0 \,..\, 2^{\ell_{\mathsf{scalar}}} - 1\})$ from $\mathsf{v}' \,||\, \mathsf{N}$
   · If the extraction fails, return $\bot$
   · Let $\mathsf{g}'_{\mathsf{d}\,\mathbf{n}} = \mathsf{DiversifyHash}(\mathsf{d}'_{\mathbf{n}})$
   · If $\mathsf{g}'_{\mathsf{d}\,\mathbf{n}} = \bot$, return $\bot$
   · Let $\mathsf{cm}'_{u\,\mathbf{n}} = \mathsf{Extract}_{\mathbb{J}^{(r)}}(\mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}'_{\mathbf{n}}}(\mathsf{repr}_{\mathbb{J}}(\mathsf{g}'_{\mathsf{d}\,\mathbf{n}}), \mathsf{repr}_{\mathbb{J}}(\mathsf{pk}'_{\mathsf{d}\,\mathbf{n}}), \mathsf{v}'_{\mathbf{n}}))$
   · If $\mathsf{cm}'_{u\,\mathbf{n}} \neq \mathsf{cm}_{u\,\mathbf{n}}$, return $\bot$

2. **Value commitment randomness verification.**

   As with Mint transfers, vault needs to know the trapdoor used to generate the value commitment cv in the Burn transfer in order to be able to submit proofs of balance including said transfer. Furthermore, they need to know the burned value, as they cannot derive it from the released value due to the rounding error in calculating the latter.

   Hence they verify that the value commitment in the burn transaction was derived from $\mathsf{rcm}_{\mathbf{n}}$ and the transmitted value as follows:

   · Let v' and $\mathsf{rcm}_{\mathbf{n}}$ be the values extracted from $\mathsf{N^{enc}}$
   · Let $\mathsf{rcv}' = \mathsf{LEOS2IP}_{256}(\mathsf{BLAKE2b\text{-}256}(\text{“}\texttt{ZCLAIM\_derive\_cv}\text{”}, \mathsf{rcm}_{\mathbf{n}})) \mod 2^{251}$
   · Let $\mathsf{cv}' = \mathsf{ValueCommit}_{\mathsf{rcv}'}(\mathsf{v}')$
   · Then, return $\top$ if $\mathsf{cv} = \mathsf{cv}'$, otherwise return $\bot$

If any of these steps fails, vault should execute *challengeRedeem* instead.

vault waits until the burn transaction has been securely included on the issuing chain.

Transaction The release transaction that vault must submit on Zcash may be any transaction in which the decrypted note $\mathbf{n}$ constructed by redeemer is the output note of an Output transfer. We call this Output transfer the

release transfer. The release transaction, as lock transactions, may contain other inputs and outputs.

Requirements The lock transaction succeeds if it follows the standard Zcash protocol rules.

Outcome This transaction has no immediate effect on I.

5.11.3 *confirmRedeem*

Pre-execution steps VAULT waits until the release transaction has been securely included on Zcash and confirmed by the relay system.

Transaction VAULT submits a confirmRedeem transaction on I, which takes $(\mathsf{cm}_{u\,\mathbf{n}}, \mathsf{pos}_{\mathbf{n}}, \mathsf{rt}_{\mathbf{n}}, \mathsf{path}_{\mathbf{n}})$ as input, where:

- $\mathsf{cm}_{u\,\mathbf{n}} : \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]}$ must be the note commitment hash in the Burn transfer, which is at the same time that of the released note.
- $\mathsf{pos}_{\mathbf{n}} : \{0\,..\,2^{\mathsf{MerkleDepth}^{\mathsf{Sapling}}} - 1\}$ must be the index of the leaf node occupied by the hash value $\mathsf{cm}_{u\,\mathbf{n}}$ in Zcash's note commitment tree.
- $\mathsf{rt}_{\mathbf{n}} : \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]}$ must be an anchor for the output treestate of a block of block height $h \geq h_{\mathbf{n}}$, where $h_{\mathbf{n}}$ is the height of the block in which $\mathbf{n}$ was created.
- $\mathsf{path}_{\mathbf{n}} : (\mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]})^{[\mathsf{MerkleDepth}^{\mathsf{Sapling}}]}$ must be a Merkle path from the leaf node containing $\mathsf{cm}_{u\,\mathbf{n}}$ to $\mathsf{rt}_{\mathbf{n}}$, generated as described in [17, Section 4.8].

Effectively, this transaction shows that $\mathsf{cm}_{u\,\mathbf{n}}$ exists in Zcash's note commitment tree, proving that VAULT has created the requested note. There is no signature in confirmRedeem transactions since indeed any participant may submit them, which only adds to the reliability of the protocol.

Requirements The confirmRedeem transaction succeeds if $\mathsf{cm}_{u\,\mathbf{n}}$ is the requested note commitment hash in a pending burn transaction and $\mathsf{path}_{\mathbf{n}}$ can be verified to be a valid Merkle path from $\mathsf{cm}_{u\,\mathbf{n}}$ at position $\mathsf{pos}_{\mathbf{n}}$ to $\mathsf{rt}_{\mathbf{n}}$, as specified in the success requirements of *mint* operations in Section 5.10.3.

Outcome The burn transaction is confirmed and VAULT can deduct the burned value from future proofs of balance.

accepts_redeem is set to $\top$ for VAULT and the ZEC obligations value commitment cb associated with VAULT is updated as follows:

· Let cv be the ICZ value commitment in the Burn transfer

· Let '+' be the abelian group operation on public keys defined in [17, Section 4.1.6.2] as instantiated for RedJubjub; '-cv' such that $\mathsf{pk} + -\mathsf{pk} = \mathcal{O}_+$, the group identity; and $\mathsf{pk}_1 - \mathsf{pk}_2 = \mathsf{pk}_1 + -\mathsf{pk}_2$
· Then, let $\mathsf{cb}^{\mathsf{new}} = \mathsf{cb}^{\mathsf{old}} - \mathsf{cv}$

The warranty collateral ICN_w is returned to REDEEMER.

5.11.4 *challengeRedeem*

The purpose of this operation is the same as that of a *confirmIssue* operation, i.e. to reveal the encrypted ciphertext such that it can be verified that its content is corrupt. However, in this case the goal is not to protect the contesting party from a potential loss of funds, but to demonstrate that a request has not been constructed correctly.

Pre-execution steps VAULT has established the invalidity of the securely included pending Burn transfer as per the pre-execution steps of *confirmRedeem*.

Transaction VAULT submits a *challengeRedeem* transaction on I, which takes $(\mathsf{sharedSecret}, \mathsf{d}, \mathsf{pk_d}, \mathsf{epk}, \pi_{\mathsf{ZKChallenge}})$ as input, where:

- $\mathsf{sharedSecret} : \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{SharedSecret}$ is the shared secret presumably used by REDEEMER to encrypt $\mathsf{N}^{\mathsf{enc}}$. This value is computed by VAULT as $\mathsf{sharedSecret} = \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Agree}(\mathsf{ivk}, \mathsf{epk})$.
- d, $\mathsf{pk_d}$, epk must be the values in the Burn transfer.
- $\pi_{\mathsf{ZKChallenge}} : \pi_{\mathsf{ZKChallenge}}.\mathsf{Proof}$ is a Groth16 zk-SNARK for a Challenge statement as described for challengeRedeem transactions in Section 5.11.4, with primary inputs $(\mathsf{sharedSecret}, \mathsf{DiversifyHash}(\mathsf{d}), \mathsf{pk_d}, \mathsf{epk})$.

Requirements The challengeRedeem transaction succeeds if:

- The values d, $\mathsf{pk_d}$, epk match those in a pending Burn transfer.

  As with challengeIssue transactions, the values $(\mathsf{d}, \mathsf{pk_d})$ are enough to uniquely identify the Burn transfer. If the protocol is expanded to allow multiple concurrent instances of Redeem per vault, a mechanism should be put in place in order to uniquely associate challengeRedeem transactions with Redeem transfers.
- The verification of $\pi_{\mathsf{ZKChallenge}}$ as described in [17, Appendix B.2] succeeds.
- Any of **Note commitment verification** or **Value commitment randomness verification** as defined in the pre-execution steps of *release* fail.

  This can be verified using the revealed shared secret as follows:

- Attempt decryption and verification of $\mathsf{N}^{\mathsf{enc}}$ as described in **Note commitment verification**. Skip step 1 and let sharedSecret be the value revealed by VAULT in this transaction.
- If the decryption/verification algorithm returns $\bot$, return $\top$.
- Verify that the trapdoor for cv was correctly derived as described in **Value commitment randomness verification**.
- If the verification algorithm returns $\bot$, return $\top$. Otherwise return $\bot$.

Outcome The burn transaction is discarded and REDEEMER's warranty collateral ICN_w is transferred to VAULT.

accepts_redeem is set to $\top$ for VAULT.

5.12 Balance statements

A vault VAULT needs to periodically prove statements regarding their ZEC obligations, mostly with respect to the amount of collateral they have locked. Operations proving such statements are described in this section.

5.12.1 *submitPOB*

Proofs of balance are the most essential balance statement that vaults need to provide. They prove that vaults are properly collateralised, i.e. that they have enough collateral to back their ZEC obligations at the collateralisation ratio σ_{std} as defined in Section 5.8.

Pre-execution steps VAULT cannot or does not wish to accept further lock transactions.

Transaction A submitPOB transaction takes $(\mathsf{d}, \mathsf{pk_d}, \mathsf{cb}, \mathsf{coll}, \mathsf{xr}_n, \pi_{\mathsf{ZKPOB}})$ as input, where:

- $(\mathsf{d}, \mathsf{pk_d})$ is the diversified payment address associated with VAULT in the vault registry.
- cb is VAULT's balance commitment, i.e. the value commitment to the ZEC obligations associated with VAULT in the vault registry.
- $\mathsf{coll} : \{0 \,..\, \mathsf{coll}_{\mathsf{max}}\}$ is the amount of collateral for which VAULT wishes to provide a POB. If this value is smaller than the current amount of collateral VAULT has locked, the difference is released to VAULT.
- $\mathsf{xr}_n : \{0 \,..\, \mathsf{xr}_G\}$ is the exchange rate nominator, where 1 ZEC = $\frac{\mathsf{xr}_n}{\mathsf{xr}_G}$ ICN, of an exchange rate less than or equal to the latest provided by $\mathcal{O}_{\mathsf{xr}}$.

- $\pi_{\mathsf{ZKPOB}} : \pi_{\mathsf{ZKPOB}}.\mathsf{Proof}$ is a Groth16 zk-SNARK for a POB statement, which proves that, given a primary input

$$\begin{aligned}(&\mathsf{coll} : \{0 \,..\, 2^{\ell_{\mathsf{value}^I}} - 1\},\\ &\mathsf{xr}_n : \{0 \,..\, 2^{\ell_{\mathsf{xr}_n}} - 1\},\\ &\mathsf{cb} : \mathsf{ValueCommit.Output})\end{aligned}$$

the prover knows an auxiliary input

$$\begin{aligned}(&\mathsf{b} : \{-\frac{r_{\mathbb{J}} - 1}{2} \,..\, \frac{r_{\mathbb{J}} - 1}{2}\},\\ &\mathsf{rem} : \{0 \,..\, 2^{\ell_{\mathsf{scalar}}} - 1\},\\ &\mathsf{rcb} : \{0 \,..\, 2^{\ell_{\mathsf{scalar}}} - 1\})\end{aligned}$$

such that the following conditions hold:

Collateral coverage check $\mathsf{b} \cdot \mathsf{xr}_n \cdot p_{\sigma_{std}} = \mathsf{coll} \cdot \mathsf{xr}_G \cdot q_{\sigma_{std}} - \mathsf{rem}$

Balance commitment integrity $\mathsf{cb} = \mathsf{ValueCommit}_{\mathsf{rcb}}(\mathsf{b})$

vault must instantiate this zk-SNARK with primary inputs as described above and auxiliary inputs calculated as follows:

  - b is the amount of ZEC obligations associated with vault. This value can be obtained as follows:
    - For all lock transactions sent to vault, calculate the minted value as $\mathsf{v}_{\mathsf{m}i} = \lfloor \mathsf{v}_{\mathbf{n}i} \cdot (1 - f) \rfloor$, where $\mathsf{v}_{\mathbf{n}i}$ is the value of the i-th lock transaction and $0 \leq i \leq n$
    - For all release transactions sent by vault, let $\mathsf{v}_{\mathsf{b}j}$ be the transmitted burned value in the corresponding j-th burn transaction, where $0 \leq j \leq m$
    - Then, $\mathsf{b} = \sum_{i=1}^{n} \mathsf{v}_{\mathsf{m}i} - \sum_{j=1}^{m} \mathsf{v}_{\mathsf{b}j}$
  - rem can be easily computed as $\mathsf{rem} = \mathsf{coll} \cdot \mathsf{xr}_G \cdot q_{\sigma_{std}} - \mathsf{b} \cdot \mathsf{xr}_n \cdot p_{\sigma_{std}}$.
  - rcb is obtained similarly to b using the value commitment trapdoors of Mint and Burn transfers. vault can calculate this value as follows:
    - For all confirmed Mint transfers, let $\mathsf{rcv}_{\mathsf{m}i}$ be the value commitment trapdoor as calculated by vault in **Value commitment randomness verification** as described in Section 5.10.4, where $0 \leq i \leq n$
    - For all confirmed Burn transfers, let $\mathsf{rcv}_{\mathsf{b}j}$ be the value commitment trapdoor as calculated by vault in **Value commitment randomness verification** as described in Section 5.11.2, where $0 \leq j \leq m$

· Then, $\mathsf{rcb} = \sum_{i=1}^{n} \mathsf{rcv}_{\mathsf{m}i} + \sum_{j=1}^{m} -\mathsf{rcv}_{\mathsf{b}j}$, where we define '+' to be the abelian group operation on private keys defined in [17, Section 4.1.6.2] as instantiated for RedJubjub and '-sk' such that $\mathsf{sk} + -\mathsf{sk} = \mathcal{O}_+$, the group identity

The values b and rcb can be calculated incrementally and VAULT should store the computed values such as to be able to reuse them in future proofs of balance.

The condition $\mathsf{cb} = \mathsf{ValueCommit}_{\mathsf{rcb}}(\mathsf{b})$ will hold if cb, rcv and b have been computed correctly due to the homomorphic properties of ValueCommit, with the same reasoning as argued in the context of fee calculation integrity in Section 5.10.3 and also explained in detail in [17, Section 4.12].

In essence, this zk-SNARK proves that

$$(\sum_{i=1}^{n} \mathsf{v}_{\mathsf{m}i} - \sum_{j=1}^{m} \mathsf{v}_{\mathsf{b}j}) \cdot \mathsf{xr}_V \cdot \sigma_{std} \leq \mathsf{coll}$$

holds i.e. that VAULT is properly collateralised for some particular $\mathsf{xr}_V = \frac{\mathsf{xr}_n}{\mathsf{xr}_G}$.

Requirements This transaction succeeds if:

- d, pk_d and cb are the values stored in the vault registry.
- $\mathsf{coll} \leq \mathsf{coll}'$ for coll' stored in the vault registry.
- $\frac{\mathsf{xr}_n}{\mathsf{xr}_G} \leq \mathsf{xr}_\mathcal{O}$, the latest exchange rate provided by the oracle. Some margin may be allowed in an actual implementation in order to account for exchange rate fluctuations in the delay the transaction is processed.

  This ensures VAULT is only providing an upper bound for their ZEC obligations and not revealing the actual amount by submitting a proof of balance for an artificially high exchange rate.
- π_{ZKPOB} is successfully verified.

Outcome In the vault registry, accepts_issue is set to $\perp$ for VAULT and coll and xr_n are set to the values in this transaction.

5.12.2 *submitPOC*

Proofs of capacity are considered a type of proof of balance. They allow VAULT to prove a stricter version of the statement proven in *submitPOB*, which shows that they not only have enough collateral to back their ZEC obligations, but also to accept a new lock transaction of unknown value.

Pre-execution steps VAULT wants to accept further lock transactions and has enough collateral to fulfill the constraints discussed next.

Transaction This transaction is identical to a submitPOB transaction save for the following primary and auxiliary inputs to the zk-SNARK:

- The primary input cb must be computed as $\mathsf{cb}' + \mathsf{ValueCommit}_0(\mathsf{v}_{\mathsf{max}})$, where cb' is the balance commitment associated with VAULT in the vault registry.
- The auxiliary input b is obtained in a similar way by adding $\mathsf{v}_{\mathsf{max}}$ to the value b' computed as described in *submitPOB*. Note that this also means that the value rem must be recomputed.

Requirements The success requirements for a submitPOC transaction are the same as those for a submitPOB transaction. Only the zk-SNARK must be verified using the primary input cb computed as described above.

Outcome Same as in submitPOB, but accepts_issue is set to $\top$.

5.12.3 *submitPOI*

As opposed to the two previous operations, this operations demonstrates the vault's inability to fulfill certain requests, specifically redeem requests.

Pre-execution steps VAULT does not have enough ZEC obligations to offer a redeem for $\mathsf{v}_{\mathsf{max}}$ ZEC. VAULT may continue offering to redeem, but they are not required to do so.

Transaction A submitPOI transaction takes $(\mathsf{d}, \mathsf{pk}_\mathsf{d}, \mathsf{cb}, \pi_{\mathsf{ZKPOB}})$ as input, where:

- $(\mathsf{d}, \mathsf{pk}_\mathsf{d})$ is the diversified payment address associated with VAULT in the vault registry.
- cb is VAULT's balance commitment, i.e. the value commitment to the ZEC obligations associated with VAULT in the vault registry.
- $\pi_{\mathsf{ZKPOI}} : \pi_{\mathsf{ZKPOI}}.\mathsf{Proof}$ is a Groth16 zk-SNARK for a POI statement, which proves that, given a primary input

  $$(\mathsf{cb} : \mathsf{ValueCommit.Output})$$

  the prover knows an auxiliary input

  $$\begin{aligned}(&\mathsf{b} : \{0 \,..\, \mathsf{v}_{\mathsf{max}} - 1\},\\ &\mathsf{rem} : \{1 \,..\, \mathsf{v}_{\mathsf{max}}\},\\ &\mathsf{rcb} : \{0 \,..\, 2^{\ell_{\mathsf{scalar}}} - 1\})\end{aligned}$$

such that the following conditions hold:

Insolvency check $\mathsf{b} + \mathsf{rem} = \mathsf{v}_{\mathsf{max}}$

Balance commitment integrity $\mathsf{cb} = \mathsf{ValueCommit}_{\mathsf{rcb}}(\mathsf{b})$

Requirements The submitPOI transaction succeeds if d, pk_{d} and cb are the values stored in the vault registry and π_{ZKPOB} is successfully verified.

Outcome The variable accepts_redeem is set to $\perp$ for the vault with associated diversified payment address $(\mathsf{d}, \mathsf{pk}_{\mathsf{d}})$ in the vault registry.

5.12.4 *rebalance*

This operation allows VAULT to decrease their ZEC obligations if they find themselves in a situation in which this is necessary, such as due to an increase in the exchange rate or if they wish to accept new lock requests but their collateralisation ratio is too low to submit a proof of capacity. This is not actually a balance statement, but it is unrelated to the Issue and Redeem protocols and concerns a vault's balance, hence it is listed in this section.

Pre-execution steps VAULT acquires ICZ by swapping them for ICN, acquiring them on an exchange or any other means.

Transaction In its structure, a rebalance transaction is similar to a burn transaction. It is also composed of a number of Sapling inputs and outputs and a modified Output transfer, in this case a Rebalance transfer. A Rebalance transfer is a stripped-down version of a Burn transfer, which takes as input $(\mathsf{cv}, \mathsf{d}, \mathsf{pk}_{\mathsf{d}})$ where:

- cv must be a value commitment to the ICZ value v being burned, where $\mathsf{v} : \{0 \,..\, \frac{r_{\mathbb{J}}-1}{2}\}$.
- $(\mathsf{d}, \mathsf{pk}_{\mathsf{d}})$ is VAULT's diversified payment address.

Requirements The rebalance transaction succeeds if a vault with diversified payment address $(\mathsf{d}, \mathsf{pk}_{\mathsf{d}})$ exists in the vault registry. Besides, it must be constructed according to the standard Sapling protocol rules. In particular, the Binding Signature as defined in [17, Section 4.12] guarantees that the value being burned exists on I before the transaction takes place.

Outcome The burned value is subtracted from VAULT's ZEC obligations by updating the value commitment cb associated with VAULT as in confirmRedeem transactions. See Section 5.11.3 for details.

5.13 Fee policy

Zclaim participants incur different types of fees for every transaction. We differentiate between Zcash transaction fees, transaction fees on I and Zclaim transaction fees.

5.13.1 Blockchain fees

We denote by Zcash transaction fees and transaction fees on I the inherent cost of transactions on each blockchain, which are unrelated to Zclaim but must still be paid by participants. The default transaction fee on Zcash is 0.0001 ZEC, which is less than one USD cent at the time of writing. We deem this fee small enough to be omitted in calculations and only briefly take it into consideration when discussing sending a large number of Zcash transactions, in the context of the splitting strategy conceived in the next section.

We assume transaction fees on I to also be small enough to be omitted, though another issue arises concerning these fees: the currency in which they are to be paid. A complete implementation of Sapling on I would imply fees may be paid in ICZ on a per-transaction basis, as in Zcash. However, non-monetary Zclaim transactions such as requestLock or confirmIssue transactions, and furthermore warranty collateral payments, cannot be effectuated in ICZ, as the sender of such a transaction may not own any ICZ. Hence, the fees for these transactions must be paid in ICN. We solve the immediate concern that this raises regarding privacy by arguing that, Sapling being implemented on I, ICN can also be sent through a shielded pool, hence hiding the owner's identity. Alternatively, a proxy service can be used that forwards user transactions for a small fee, as suggested by the Tezos development team concerning this very issue [41].

5.13.2 Zclaim fees

Zclaim transaction fees are the fees paid to vaults in the context of issuing and redeeming.

Vaults must derive a fee for their services in order to ensure a large enough number of participants assume this role, hence ensuring the security of the protocol. This fee is currently set to $f = f_n / f_G$ of the transacted ICZ amount in every successful Issue or Redeem procedure in which they are involved.

Specifically, in Issue procedures vaults receive some value $\mathsf{v_n}$ locked on Zcash, but only the minted value $\mathsf{v} = \lfloor \mathsf{v_n} \cdot (1 - f) \rfloor$ is added to their ZEC obligations. Hence they receive $\mathsf{v_n} - \mathsf{v} = \lceil \mathsf{v_n} \cdot f \rceil$ ZEC more than the amount they need to back with ICN, which is their fee.

In Redeem procedures on the other hand, the burned value v is subtracted from the vault's ZEC obligations, whereas they only need to release $v_{\mathbf{n}} = \lfloor v \cdot (1 - f) \rfloor$. Hence, in this case, they release $v - v_{\mathbf{n}} = \lceil v \cdot f \rceil$ less than the amount subtracted from their ZEC obligations. Note that this really is the same fee for both protocols: there is no difference between receiving ZEC and reducing ZEC obligations, since ZEC obligations eventually need to be released 1-to-1 for ZEC.

This fee is currently the same for both protocols for simplicity, though some interoperability solutions offer a different fee to redeem than to issue [56]. There is nothing that would prevent this from being implemented on ZCLAIM if it was desired. Furthermore, it would be possible to let vaults individually set a fee and to advertise them along with this fee, which is also currently not implemented for simplicity.

5.14 Splitting strategy

In order to prevent vaults from learning the amounts issued and redeemed through them, we devise a strategy to split the total amount into separate transactions such that vaults cannot know how much is actually being transacted. See Sections 6.2.1 and 6.3 for the reasoning behind this functionality.

In a protocol with n vaults available to issue, the total v_{tot} is split into k (where possibly $k > n$) parts such that each of the resulting amounts x_1, x_2, $x_3 ... x_k$ is issued separately, and analogously for redeeming.

The problem can be defined as finding a suitable approach such that the amount of information vaults learns through knowledge of a number of these values is minimised or decreased as far as possible.

5.14.1 Related work

Zcash Sprout-Sapling migration approach

Incidentally, a very similar problem has been faced in the past by the Zcash development team, with the aim of migrating funds between addresses in two different versions of the protocol, Sprout and Sapling [59, 60]. The Zcash consensus rules prohibit direct transfers from Sprout to Sapling shielded addresses, unless the amount is revealed by sending it through the 'transparent value pool', which involves crafting a shielded-to-transparent and a subsequent transparent-to-shielded transactions. This represents a privacy risk to the users, in the same way locking funds with a vault does in our case.

Their approach involves creating multiple transfers picked according to a random distribution. Up to 5 of these transactions are made whenever the

blockchain reaches a 500 block (∼10 hours) height interval. This means that the migration takes a considerable amount of time even for smaller amounts, relatively speaking.

The details of the protocol can be found under [60]. In short, the amount is chosen by sampling an exponent and a mantissa each from a uniform distribution and then sending the resulting amounts with base 10 through the transparent value pool. By limiting the frequency of transactions and collecting those from all migrations happening in the same block it is possible to leak less information about the distribution of amounts, though no formal analysis is made as to how much information this is.

Applicability Though the problem being addressed is very similar in both cases, there are some significant differences that must be highlighted.

First of all, the migration leverages the fact that it does not need to be completed within a particular timeframe. Migrating a large amount might take up to several months, which would be unacceptable in our case when what we are trying to achieve are efficient exchanges. It is not clear why the migration parameters determining the pace were set to these specific values (batches of 5 transactions, intervals of 500 blocks).

Furthermore, this approach was chosen under the assumption of a global observer, as all transactions are public and observable by anyone. In our protocol, however, we may assume that only a fraction of all vaults are controlled by an adversary and hence they only have knowledge of a subset of all transactions happening. Thus the security assumptions of this strategy may be too strict for our purposes.

Mixicles approach

In their *Mixicles: Simple Private Decentralized Finance* paper, Juels et al. [61] describe a method that provides perfect sub-transaction privacy against a global observer with knowledge of the total amount. This is of course a different situation than in our case, where the opposite applies: the observer has knowledge of a fraction of all transactions happening, and we would like to avoid them learning the total amount. However, their work is still of interest to us.

Let us first describe their protocol more precisely. The relevant components are: a total amount t, known by the adversary, and two payout amounts p_0 and p_1 that are transferred by a set of transactions to two recipients. The amount of each transaction is public, but not its recipient.

The total t can be split into a set D of size $\log_2 t$ of smaller amounts in such a way that any amount smaller than t can be represented by a number of elements in this set. This means that both payments p_0 and p_1 can be made

with this set of transactions by only altering the recipients, which are private. Thus regardless of the payout split, no excess information is revealed.

Applicability This approach is inapplicable to our case since the construction of this set is deterministic depending on the amount, which means that knowledge of only one term may be sufficient to deduce the total amount.

A similar approach could however be conceived: use a fixed set for all transactions with which it is possible to represent any number (powers of a base), and send elements of this set to different vaults.

With this approach, the observer will learn very little information about the total amounts being sent. In cases where an amount equal or larger to the largest multiple is being sent, one vault will learn that this is the case, but still not be able to guess the total with reasonable certainty.

The only problem this poses is the information that the number of transactions reveals about the total, which may pose a problem under low network traffic.

5.14.2 Base representation

We expand on the approach briefly described above.

Given the total amount $\mathsf{v}_{\mathsf{tot}}$, we proceed as follows:

1. Choose a base $b \in \mathbb{N}$.
2. Construct v_b, the representation of $\mathsf{v}_{\mathsf{tot}}$ in base b.
3. For every digit n_i in v_b, send n_i transactions of value b^i, where $i_{\mathsf{min}} \leq i \leq i_{\mathsf{max}}$ such that $b^{i_{\mathsf{min}}} = \mathsf{v}_{\mathsf{min}}$ for some minimum transaction value $\mathsf{v}_{\mathsf{min}}$ and $b^{i_{\mathsf{max}}} = \mathsf{v}_{\mathsf{max}}$ (implying we set $\mathsf{v}_{\mathsf{min}}$ and $\mathsf{v}_{\mathsf{max}}$ such that these relations hold).

Users may always choose a maximum exponent $i_{maxU} < i_{\mathsf{max}}$ if they wish to conceal larger amounts and send several transactions of value $b^{i \leq i_{maxU}}$ instead.

Base 2 leads to the lowest number of transactions on average assuming all values for $\mathsf{v}_{\mathsf{tot}}$ in the interval $[\mathsf{v}_{\mathsf{min}}, \mathsf{v}_{\mathsf{max}}]$ are allowed, but in principle this can be accomplished with any base. We choose $b = 2$ since we see no clear benefits inherent in using a larger base.

5.15 Liquidation

Liquidation involves the public sale of a fraction or all of a vault's collateral at a favourable exchange rate, thus penalising the vault. The discrepancy

between the actual exchange rate and that used in liquidation auctions is called the *liquidation penalty*. Liquidation may be triggered by a number of different factors:

- If a vault has failed to react to some fixed number of issue or redeem requests. For example, we may say that the third time a vault fails to confirm a burn transaction, they are liquidated by the equivalent of $\mathsf{v}_{\mathsf{max}}$ in ICN at the liquidation markdown. The liquidated amount may for some short amount of time only be offered to the redeemer whose request was neglected, and only after that to the public.
- If the exchange rate increases and the vault fails to provide a proof of balance that reflects the change in the exchange rate.

  Liquidation on vault V is triggered at time T_{liq} if

  $$\mathsf{xr}_{T_{liq}} > \mathsf{xr}_V \cdot \frac{\sigma_{std}}{\sigma_{min}}$$

  where xr_V is the exchange rate used as parameter in V's last POB/-POC, σ_{min} is the minimum collateralisation ratio and σ_{std} is the standard collateralisation ratio. Plugging in the values suggested for $\sigma_{std} = \frac{3}{2}$ and $\sigma_{min} = \frac{9}{8}$ in Section 5.8, liquidation takes place thus whenever

  $$\frac{\mathsf{xr}_{T_{liq}}}{\mathsf{xr}_V} > \frac{\sigma_{std}}{\sigma_{min}} = \frac{4}{3} \approx 1.33$$

  i.e. if the exchange rate increases by 33% with respect to xr_V.

One issue with this approach is that not only does it trigger liquidation on the same threshold regardless of the vault's actual ZEC obligations, it also must always sell the same proportion of a vault's collateral: enough to guarantee that the vault meets σ_{std} again. With the suggested value, the proportion sold is $\frac{\sigma_{min}}{\sigma_{std}} = 3/4$ of a vault's entire collateral[7], regardless of its actual ZEC obligations. This can result in a negative ZEC obligations, which results in debt to the vault's name that can be repaid by issuing.

There is currently no transaction facilitating auction bids, but we can consider what this transaction may look like. The ICZ value paid by bidders will be immediately burned in order to reduce the vault's ZEC obligations. Hence we can adapt Burn transfers to this end: if the bidder provides his ICN address in the transaction, the purchased amount can be credited to them. Further, if we open the value commitment, give liquidation auctions an identifier and also add that identifier to the transaction, we have what we needed.

[7] It may be a good idea to raise σ_{std} in an actual implementation, which will decrease this fraction and at the same time push the liquidation threshold further away.

5.16 Rebalancing

We call rebalancing the process that vaults can undertake in order to decrease their ZEC obligations. There must be a way for them to do so, since otherwise they may not be able to withdraw their collateral once they wish to stop taking part in the protocol. The process is simple:

1. *Acquire ICZ*. Remember that we assume swaps between ICN and ICZ are implemented on the issuing chain, hence if there is enough supply the vault can obtain ICN this way.

   Otherwise, they may be able to buy it on an exchange or can even Mint through another vault, in which case they will of course have to pay the fee.

2. *Rebalance.* Execute *rebalance* as described in Section 5.12.4.

Chapter 6

Analysis

In this section, we analyse ZCLAIM from different perspectives. In Section 6.1, we show that the protocol fulfills certain properties under which we consider it to be secure.

In Section 6.2, we present several common attack vectors on blockchain protocols and discuss their (in-)feasibility on ZCLAIM, providing threshold parameters guaranteeing the security of the system where appropriate.

Finally, in Section 6.3, we analyse privacy concerns related to the vault intermediary system.

6.1 Security analysis

In this section, we present a set of properties under which ZCLAIM is considered secure. We specify the conditions that need to hold in order to attain each of these properties. Finally, we show that ZCLAIM satisfies these conditions.

We presuppose the proper choice of security parameters k^I, k^B, Δ^I and Δ^B as discussed in Section 5.1 and the security of the Zcash/Zerocash protocol, which has been extensively shown in [15]. We also assume that the relay system is secure and that if a block header is accepted by the relay system, the possibility of a future chain reorganisation is vanishingly low.

6.1.1 Notation

We denote any participant in the protocol by $P \in \mathcal{P}$, who may be a vault $V \in \mathcal{V}$ or a user $U \in \mathcal{U}$ taking on the role of issuer or redeemer. The amount of currency X of any of the backing currency Zcash (ZEC), the issuing chain's native currency (*ICN*) and the issued currency (*ICZ*) that a participant can spend is denoted by $X(P)$, e.g. $ICN(U)$, to which we may refer to as U's

ICN *balance*. Specifically, $ZEC(P)$ is the sum of the value of all unspent Zcash notes of which P has knowledge containing $(\mathsf{d}, \mathsf{pk_d})_P$, the diversified payment address associated with P. Same goes for ICZ. Funds locked as collateral by participant P are denoted by $X^{col}(P)$.

In addition to the aforementioned balances, $ZEC^{obl}(V)$ are the *ZEC obligations* associated with vault V and is defined as the sum of the ZEC value of all (hidden) amounts of ICZ issued or redeemed in mint and burn transactions facilitated by V. This also represents the amount that vaults can release to redeemers and does not reflect their ZEC balance, since $ZEC(V)$ includes the fees accumulated by V in past transactions while $ZEC^{obl}(V)$ does not. Furthermore, it is not possible to verify that $ZEC(V)$ matches $ZEC^{obl}(V)$ in any way as V may perform arbitrary shielded transactions at any time.

Upon submission of a valid POB (or similarly for POCs), in which a vault proves in zero knowledge that $c \geq ZEC^{obl}(V) \cdot \mathsf{xr} \cdot \sigma_{std}$ where $c \leq ICN^{col}(V)$, V is said to have some *blocked collateral* $ICN^{bcol}(V) = ZEC^{obl}(V) \cdot \mathsf{xr} \cdot \sigma_{std}$ and some free collateral $ICN^{fcol}(V) = c - ICN^{bcol}(V)$. The vault's blocked collateral are the funds backing their ZEC obligations, which they can only decrease by releasing funds to users (or by burning ICZ, see discussion on rebalancing in Section 5.16).

$X_t(P)$ denotes P's balance at time t, where one step in time corresponds to a transaction on I. $ZEC_t(P)$ is not defined.

Finally, xr_t denotes the latest ZEC to ICN exchange rate as provided by the exchange rate oracle $\mathcal{O}_{\mathsf{xr}}$ at time t, where 1 ZEC = xr_t ICN. We assume a 1:1 peg between the issued currency and the backing currency, i.e. 1 ZEC = 1 ICZ.

6.1.2 Goals

We deem Zclaim *secure* if it achieves the following properties:

- **Soundness** The amount of issued currency in circulation is equal to the amount of ZEC obligations, i.e. for all $t \geq 0$

$$\sum_{U \in \mathcal{U}} ICZ_t(U) = \sum_{V \in \mathcal{V}} ZEC_t^{obl}(V) \tag{6.1}$$

  and $ZEC^{obl}(V)$ is derived from protocol transactions to and from V on Zcash. Specifically,

  Lemma 6.1 *For every Mint transfer of (hidden) value v_{IM} on I containing $(\mathsf{d}, \mathsf{pk_d})_V$ as the vault's diversified payment address, there is an Output transfer on Zcash creating a note of value v_Z to $(\mathsf{d}, \mathsf{pk_d})_V$ s.t.*

$$\lfloor \mathsf{v}_Z \cdot (1 - f) \rfloor = \mathsf{v}_{IM} \tag{6.2}$$

and

Lemma 6.2 *For every Burn transfer of value* v_{IB} *on I containing* $(\mathsf{d}, \mathsf{pk_d})_V$ *as the vault's diversified payment address and* cm_Z *as the requested release note commitment, there is an Output transfer on Zcash creating a note of value* v_Z *with note commitment* cm_Z *s.t.*

$$\mathsf{v}_Z = \lfloor \mathsf{v}_{IB} \cdot (1 - f) \rfloor \tag{6.3}$$

according to the fee policy defined in Section 5.13.

These are the transfers that make up a vault's ZEC obligations, and evidently the sum of all Mint transfers minus all Burn transfers is equal to the circulating supply. Hence Eq. (6.1) is in fact satisfied by the definition of $ZEC^{obl}(V)$.

- **Coverage** The total amount of ZEC obligations are backed by a proportional amount of the issuing chain's native currency according to the prevailing exchange rate, i.e. for all $t \geq 0$

$$\left(\sum_{V \in \mathcal{V}} ZEC_t^{obl}(V) \right) \cdot \mathsf{xr}_t \cdot \sigma_{min} \leq \sum_{V \in \mathcal{V}} ICN_t^{bcol}(V) \tag{6.4}$$

  where σ_{min} is the minimum collateralisation ratio as defined in Section 5.8.

- **Fairness** An honest participant following best practices will not incur any loss of funds as long as they can receive and broadcast transactions from and to chains Z and I.

  For the condition on this claim, availability of both chains must be guaranteed, which we derive from the security assumption of XCLAIM that transactions broadcast by users are received by (honest) consensus participants within a known maximum delay Δ_{tx}. In case of network failure for a significant amount of time on the user/vault side, they may in effect incur loss of funds. It is thus the participants' responsibility to ensure they remain online throughout the duration of a subprotocol.

  As for the claim itself, we examine the cases where a participant may incur loss of funds. This risk exists both in the issue and redeem subprotocols. We concern ourselves with transactions that incur a change in any of the balances of the two parties involved (except for slashing, which we cover separately) i.e. in which a monetary transaction takes place. These are, on the one hand, lock and release transactions, and on the other, mint and burn transactions.

  Concerning the issue subprotocol, the following statements must be proven:

Lemma 6.3 *After executing a lock operation, a user is able to mint the locked amount minus fees of ICZ.*

Lemma 6.4 *If a* mint *transaction involving a vault V is confirmed on I (thus increasing their ZEC obligations), V has received the amount being minted plus fees in a previous* lock *transaction.*

Note that Lemma 6.4 is equivalent to Lemma 6.1 with the added requirement that V must have knowledge of the note values.

Analogously, for redeeming we must show that:

Lemma 6.5 *After executing a release operation, a vault is able to trigger the inclusion of an associated pending* burn *transaction decreasing their ZEC obligations by the released amount plus fees.*

Lemma 6.6 *If a* burn *transaction involving a user U is confirmed on I, U has received the amount being burned minus fees in ZEC in a previous* release *transaction.*

Again, Lemma 6.6 is tantamount to Lemma 6.2 with addition of the knowledge requirement.

Consideration must additionally be paid to the slashing mechanism, in order to ensure that slashing of an honest party cannot be instigated by a malicious one.

Finally, we provide a set of practices that vaults may follow in order to prevent liquidation and hence loss of funds.

6.1.3 Argumentation

We show that Lemmas 6.1 and 6.2 are always satisfied and, furthermore, that the recipients of the Zcash notes referenced therein have knowledge of the note values, hence satisfying Lemmas 6.4 and 6.6. Further, we cover Lemmas 6.3 and 6.5, and show the validity of Eq. (6.4). Finally we discuss slashing and liquidation.

Issuing

The zk-SNARK π_{ZKMint} in Mint transfers as defined in Section 5.10.3 guarantees that Lemma 6.1 holds through the **Note commitment integrity** condition together with the Merkle path validity success requirement (there exists such a note on Zcash), and the **Locked value**, **Minted value** and **Fee rounding commitment integrity** conditions (the note has the specified value).

Mint transfers also satisfy Lemma 6.3 through **Trapdoor commitment integrity**, which guarantees that only the user that authored the lock transaction can create a mint transaction.

Furthermore, the challenge mechanism guarantees that the vault has knowledge of the note values, satisfying Lemma 6.4, since mint transactions will only be included on I if they are not successfully challenged. The vault can challenge the pending transaction through disclosure of the shared secret, proven to be correct in $\pi_{\mathsf{ZKChallenge}}$. If the note values have not been correctly encrypted to the vault, the transaction will be discarded, which ensures **Fairness**.

So far we have proven Lemmas 6.1, 6.3 and 6.4.

Redeeming

Equally, the conjunction of π_{ZKBurn} as defined in Section 5.11.1 and the confirmRedeem transaction guarantees that Lemma 6.2 holds through the conditions in π_{ZKBurn} and the Merkle path validity success requirement of confirmRedeem transactions.

Lemma 6.5 is rather trivial as all it takes is to provide a Merkle path showing the existence of the note commitment in the note commitment tree, which not only the vault but any participant can do.

As for redeeming, the challenge mechanism works somewhat differently: it is the recipient of the note (the redeemer) who constructs it, and the sender must have knowledge of the values in order to be able to create it. Thus Lemma 6.6 is satisfied by construction, and we can move on to the challenge mechanism: in this case, it serves to ensure the vault has received the correct note values.

A vault may successfully challenge the transaction if and only if the provided note commitment cannot be generated from the decrypted values, thus it cannot punish honest redeemers but also a redeemer cannot ask a vault to release a note it cannot create, causing it to be slashed. Hence **Fairness** is ensured when redeeming too.

Now, we have also proven Lemmas 6.2, 6.5 and 6.6, in addition to Lemmas 6.1, 6.3 and 6.4 from before, which concludes the proof for **Soundness**. It remains to prove **Coverage** and to discuss loss of funds in liquidations and challenge operations.

Proofs of balance

From Eq. (6.4) it is evident that if we can show that

$$ZEC_t^{obl}(V) \cdot \mathsf{xr}_t \cdot \sigma_{min} \leq ICN_t^{bcol}(V)$$

for any vault, then (6.4) holds for the entire system.

We recall here that $ICN^{bcol}(V) = ZEC^{obl}(V) \cdot \mathsf{xr} \cdot \sigma_{std}$, which nevertheless only holds for some xr used in the POB. We shall denote this exchange rate by xr_V.

Technically, it also only holds for $ZEC^{obl}(V)$ at the time of its last POB, though the only problematic case is that in which a vault's ZEC obligations increase since that time, i.e. they issue funds.

However, vaults may only become available to issue by submitting a POC (and must provide POCs instead of POBs until issuing is completed), in which they prove collateralisation for their ZEC obligations plus the maximum amount they can issue in one round. Thus we may assume without loss of generality that $ZEC^{obl}(V)$ stays constant.

For any $\mathsf{xr}_t > \mathsf{xr}_V$ then it must hold that

$$ZEC^{obl}(V) \cdot \mathsf{xr}_t \cdot \sigma_{min} \leq ZEC^{obl}(V) \cdot \mathsf{xr}_V \cdot \sigma_{std}$$

$$\mathsf{xr}_t \cdot \sigma_{min} \leq \mathsf{xr}_V \cdot \sigma_{std}$$

which is the liquidation trigger as defined in Section 5.15.

Liquidation reduces V's ZEC obligations until they meet σ_{std} again as described in the aforementioned section.

Thus we see that (6.4) does indeed always hold and is enforced by reducing the circulating supply if needed through liquidation auctions. This concludes the proof for the **Coverage** property.

Challenging

Challenge operations result in the challenged party's warranty collateral ICN_w being transferred to the challenging party if successful (for vaults, this amount is deducted from their staked collateral). However, by definition these are only successful if the note plaintext in the challenged transaction has not been correctly encrypted to the challenger. Thus it is not possible to cause loss of funds through these operations to an honest participant that adheres to the protocol.

Liquidation

In the event of liquidation, vaults are effectively slashed the liquidation penalty on the funds sold through the liquidation auction. The amount of funds sold depends on the collateralisation constants and is always a fixed fraction of the vault's total collateral.

As mentioned in Section 5.15, with the collateralisation constants as suggested, liquidation is triggered on a vault if the exchange rate increases by 33% with respect to xr_V, the last exchange rate they used in a proof of balance.

In order to avoid liquidation, V needs to provide a POB using a higher xr before the aforementioned threshold is reached. If at any time t where $T_p < t < T_{liq}$ and $\mathsf{xr}_V < \mathsf{xr}_t < \mathsf{xr}_{T_{liq}}$, it holds that

$$ZEC_t^{obl}(V) \cdot \mathsf{xr}_t \cdot \sigma_{std} \leq ICN_t^{col}(V) \tag{6.5}$$

the vault can submit a new POB using xr_t, hence avoiding liquidation at T_{liq}. Naturally, this holds for later steps in time: if the exchange rate further increases, the vault will need to continue providing POBs in order to protect themselves from liquidation. Vaults should choose a sensible $\mathsf{xr}_t > \mathsf{xr}_V$ for which to submit a new POB before they approach the liquidation threshold.

However, if Eq. (6.5) is no longer satisfied, i.e. the change in the exchange rate at time t has caused V's collateralisation rate to fall below σ_{std}, V first needs to top up their collateral or rebalance in order to be able to submit a POB using xr_t.

If they hold ICN such that $ZEC_t^{obl}(V) \cdot \mathsf{xr}_t \cdot \sigma_{std} \leq ICN_t^{col}(V) + ICN_t(V)$, this is straightforward (barring the case where the exchange rate further increases such that Eq. (6.5) again does not hold after collateral has been topped up) and is accomplished in the time it takes the vault to construct the two transactions and forward them to the network, which is negligible.

However, if this is not the case, V may first have to acquire some ICN to top up their collateral such that they can again submit a POB. How long it takes them to do so depends on many external factors, but it may plausibly be up to several days: they may first have to procure liquid assets, transfer them to an exchange, trade them for ICN, withdraw the ICN from the exchange and finally lock them as collateral. In this case, it may be fair to say that all is lost and vaults should follow best practices in order to avoid this sort of situation.

Another option vaults have to satisfy Eq. (6.5) is to rebalance, but this is a strictly longer process than topping up their collateral since it also requires ICN and one further transaction on I, but also exchanging ICN for ICZ. Thus if the goal is for them to protect themselves from liquidation as fast as possible, they should always choose to top up. In addition to this, vaults may choose to start rebalancing such that they do not need to top up again if the exchange rate further increases or that they may unlock some collateral again.

Best practices vaults can follow in order to protect themselves from liquidation may thus include:

- Maintaining a collateralisation rate somewhat higher than σ_{std}, i.e. overcollateralising, by strategically advertising themselves only to issue or redeem, rebalancing or increasing their collateral when necessary. For example, $\sigma_{sft} = 2$.

- Holding some amount of ICN not locked as collateral that they may use to top up their collateral or rebalance, or ensuring that they may acquire such amount quickly in case of emergency.
- Submitting POBs to address changes in the exchange rate above a certain threshold $r_{\mathsf{xr}} = \frac{\mathsf{xr}_t}{\mathsf{xr}_V}$. This should be a value close to 1.0, e.g. $r_{\mathsf{xr}} \leq 1.05$.

Vaults may modify these parameters according to their risk tolerance, but with the suggested values we observe that for liquidation to be triggered it would be necessary that $\frac{\mathsf{xr}_t}{\mathsf{xr}_V} > \frac{\sigma_{sft}}{\sigma_{min}} - r_{\mathsf{xr}} \approx 1.62$ i.e. the exchange rate increases by 62% before V is able to acquire new funds or rebalance.

Such a dramatic increase is unlikely to happen in a short time window, but in the context of cryptocurrencies, where high volatility is the norm rather than the exception [62, 63], it is impossible to set a definite threshold for the increase in the exchange rate above which the likelihood of it taking place is insignificant. Ultimately, it is left to vaults to assess the risks themselves.

Finally, liquidation may also be triggered if a vault fails to comply with issue or redeem requests too often, which should however not happen under the availability assumptions of participants under which **Fairness** applies.

Thus if we make the assumption that xr will not increase fast enough to trigger liquidation on vaults following best practices, this concludes the argumentation in favour of **Fairness**.

6.2 Attack vectors and points of failure

We discuss here a range of attacks and points of failure in Zclaim and offer mitigation strategies. Familiarity with the security analysis of Xclaim [14, Section VII] is assumed. Where not specified otherwise, the discussion on specific vulnerabilities offered there also holds for Zclaim.

6.2.1 Inference attacks

As discussed in Section 6.3, vaults may guess the users' identity through the amounts in lock and release transactions in which they are involved. The knowledge of this amount is of course per se insufficient to this end, but it can lay the ground for an inference attack leading to de-anonymisation if combined with other information they may have access to.

For instance, if an easily identifiable amount a user locks with a vault matches a recent transparent-to-shielded transaction on Zcash, the vault may deduce that there is a high likelihood those two transactions were performed by the same user. This is similar to the heuristic used by Quesnelle [64] to identify what he calls 'round-trip transactions, where the same, or

nearly the same number of coins are sent from a transparent address, to a shielded address, and back again to a transparent address. [He argues that] such behavior exhibits high linkability, especially when they occur nearby temporally' [64, p. 1]. If the vault is correct in their assumption, they may be able to infer the user's identity from previous and future activity associated with the disclosed transparent address.

Another possibility is the case where a vault has privileged access to other information on one or many users, such as through data from a cryptocurrency exchange. This may render them capable of effortlessly matching the real-world identities of exchange users with specific lock or release transactions if, for example, the guessed/observed amount matches a recent withdrawal from the exchange.

In both of these scenarios, an attacker may leverage this information to attempt extortion or defamation of the involved parties.

The splitting strategy defined in Section 5.14 aims to prevent this sort of attacks. ZCLAIM's privacy has been analysed in this context in Section 6.3.

6.2.2 Chain relay poisoning

Chain relay poisoning involves an adversary triggering a chain reorganisation, or *reorg* for short, of $N \geq k^B$ such that a previously accepted transaction at depth N is invalidated. This is known as an N-confirmation double spend and yields similarities to selfish mining [65] in that it involves misleading honest nodes through an amassing of computational power.

We recall here that the security parameter k^B is based on the assumption that an adversary's computational power is bounded by $\alpha \leq 33\%$ and denotes the depth at which the likelihood of an adversary triggering a reorg is negligible.

However, in XCLAIM a poisoning attack may be successful well below this threshold α if the relay system is deprived of recent block header data [14, Section VII-A].

We expand here on this observation. This is in fact a common vulnerability to cross-chain interoperability schemes and is not only the case if the relay system is devoid of real-world data [66], but may involve more intricate attacks in which *relayers* are isolated from the rest of their peers in the network and misled to accept the attacker's chain as the longest. This is more commonly known as an eclipse attack [67].

Such attacks along with mitigation strategies have been discussed previously in the literature [67, 68, 69, 70], and we refer the reader in particular to analyses on Bitcoin and Bitcoin-based blockchains (such as Zcash is) for mitigation strategies [67, 70].

6.2.3 Exchange rate poisoning

Similarly, if $\mathcal{O}_{\mathsf{xr}}$ is manipulated to provide an erroneous price feed, this may allow a prepared adversary to steal funds in several ways.

For instance, an exchange rate much higher than the actual value would allow vaults to issue ZEC or unlock collateral such that they become undercollateralised when the exchange rate returns to normal. An economically rational vault would have no incentive to bring back their collateralisation rate above the safety value and may choose to undergo liquidation and retain the ZEC for which they hold obligations, effectively violating **Soundness** and jeopardising the stability of the protocol.

On the other hand, an artificially low exchange rate may trigger mass liquidation and allow users to buy vaults' collateral at an unfair price, hence stealing funds from them.

It is thus of paramount importance to guarantee the reliability of the exchange rate oracle. Blockchain oracles aim to solve this exact problem [71, 72, 73], aggregating exchange rates from different sources, leveraging economic incentives to reinforce the veracity of these sources and providing dispute mechanisms in case of discrepancies. These systems are, without question, safer than relying on a single source to provide an exchange rate, though they may still fail under certain circumstances [74].

Further measures can be taken to prevent such an attack, such as sanitising data provided by the oracle or employing so-called circuit breakers [75], which involve halting operations in case of unusual price movements.

Lastly, it must be noted that relying on a decentralised oracle constitutes a further cross-chain integration and as such opens another door to poisoning attacks as discussed in the previous section, which can be addressed through the same countermeasures.

6.2.4 Replay attacks on inclusion proofs

Zclaim prevents replay attacks on lock and release transaction inclusion proofs, in which a user reuses a past lock transaction to mint ICZ or a vault reuses a release transaction to decrease their ZEC obligations, stealing funds from the other party, as follows.

The nonce $\mathsf{n}_{\mathsf{permit}}$ provided by the issuing chain in lock permits must be used to generate the note commitment trapdoor in lock transactions as specified in Section 5.10.2, which is enforced in the zero knowledge proof in Mint transfers π_{ZKMint} through the **Trapdoor commitment integrity** condition. This ensures that every lock transaction is uniquely associated with the corresponding Issue procedure.

As for release transactions, protection from replay attacks is implicit since the note commitment is generated in advance by the redeemer. Therefore a vault can only replay a release inclusion proof if the redeemer has chosen the same note values, most notably the same note commitment trapdoor rcm as in a previous Burn transfer.

However, the likelihood of the same trapdoor being sampled twice randomly from $\mathsf{NoteCommit}^{\mathsf{Sapling}}.\mathsf{GenTrapdoor}()$ as specified in Section 5.11.2 is insignificant. If a redeemer purposely reuses the same note values, they only derive negative utility from their actions.

6.2.5 Counterfeiting

In XCLAIM, counterfeiting is defined as the issuing of issued currency that is not backed by funds locked with vaults. This is enforced through the principle of **Auditability**. In a nutshell, since the vaults' actions are observable by anyone, a vault removing funds from the pool of funds locked with them can be reported and will be punished by slashing their collateral as a consequence. This also includes vaults reusing these funds to mint more issued currency.

It is here that a central difference between ZCLAIM and XCLAIM arises: we do not make such restriction. Instead, **Coverage** and **Soundness** guarantee that the total amount of issued currency in circulation is always backed by an equivalent amount of the vaults' collateral. A vault may very well reuse ZEC locked with them to issue more ICZ; they will still be unable to unlock their collateral until they have released ZEC to redeemers or burnt ICZ themselves.

Counterfeiting would hence imply issuing ICZ which is not locked by ICN, which is impossible as per the aforementioned properties.

6.2.6 Extortion

Extortion by vaults, which would involve vaults setting extreme fees to redeem hence making it unfeasible, is prevented in ZCLAIM under the suggested fee policy, which defines a fixed fee for both the issue and redeem procedures.

6.2.7 Black swan events

A black swan event is an extremely rare event with potentially catastrophic consequences for parties exposed to a previously unknown or neglected risk.

In a financial setting, this term is usually employed to describe a severe market-wide crash or extreme, sudden devaluation of an asset due to unforeseen circumstances.

Such events are a rather common occurrence in the cryptocurrency space [76, 77], perhaps challenging their definition. This is due to several factors, foremost the speculative nature of cryptocurrencies [78] and the prevailing high volatility in their valuation. Furthermore, even though advances in blockchain technology are made at a rapid pace, the technology is still in its early stages and as such is susceptible to attacks and exploits of varying nature. Finally, the regulatory framework surrounding cryptocurrencies is still being developed in many countries [79] and is a hot topic of debate [80], as it may facilitate the widespread adoption of certain types of cryptocurrencies while hindering the development of others.

This is to say that the chance of sudden, extreme devaluation of either one of Zcash or the issuing currency is non-negligible and must be kept in mind. In case of a drop in the valuation of Zcash, Zclaim would continue to operate normally and it is quite clear what the consequences would be: any participant holding either ZEC or ICZ will suffer a loss unrelated to the protocol.

If, on the other hand, the issuing currency suffered this fate, the consequences would be similar to those discussed in Section 6.2.3 and Zclaim would eventually no longer meet the security properties defined in Section 6.1.2. Zamyatin et al. [14] hence assume a minimum exchange rate xr_{min} 'below which adhering to protocol rules no longer represents the equilibrium strategy of rational adversaries' [14, p. 3], and a delay $\Delta_{\mathsf{xr}_{min}} < \Delta^I$ such that honest participants can include a transaction on I before this threshold is reached. However, it remains unclear what strategy participants may follow in order to avoid financial loss in this situation.

6.3 Privacy against vaults

We argue that the monetary values of transactions in which vaults are involved do not leak the total value prior to splitting. We also consider the information they may learn by observing network traffic and the case where several vaults collude with each other or are operated by the same adversary.

The consequences of such information leakage are discussed in Section 6.2.1.

6.3.1 Adversarial model

We assume an adversary $\mathcal{A}$ constrained by the following assumptions:

- $\mathcal{A}$ has perfect knowledge of the Zclaim protocol, i.e. also of the splitting strategy.
- $\mathcal{A}$ can observe transactions happening on both Zcash and I, but cannot filter network packets sent by individual users.

- $\mathcal{A}$ controls no more than $1/3$ of all vaults that are available to issue or redeem, independently of each other, at any given time.

6.3.2 Obfuscation of total transacted value

Using the splitting strategy defined in Section 5.14, knowledge of one or even several amounts reveals no information about the total $\mathsf{v}_{\mathsf{tot}}$ other than the evident fact that it is larger than the total observed value. Indeed, the individual amounts sent to vaults are independent of the total: the latter only determines which amounts out of the fixed set of possible values get sent. Hence, as long as $\mathcal{A}$ controls only a fraction of all vaults, the unknown amounts are equally distributed among all possible unobserved values.

However, if $\mathcal{A}$ learns the number of transactions k, this is no longer the case. We have as per the splitting strategy that $k \leq k_{max} = (b-1)(i_{max} - i_{min} + 1)$. Now, if $\mathcal{A}$ not only has knowledge of $k_{obs} < k$ amounts but also of the value $k < k_{max}$, the odds of them deducing $\mathsf{v}_{\mathsf{tot}}$ increase from $(2^{k_{max}-k_{obs}})^{-1}$ to $\left(\binom{k_{max}-k_{obs}}{k-k_{obs}}\right)^{-1}$, which is strictly larger and grows as k approaches either k_{max} or k_{obs}.

One approach to counter this issue is to split $\mathsf{v}_{\mathsf{tot}}$ into a small, random number of amounts and then apply the splitting strategy for each of these individually, such that $\mathcal{A}$ has no knowledge of k_{max} (across all amounts).

6.3.3 Transaction linkability

As we have seen, if $\mathcal{A}$ learns the number of transactions k into which $\mathsf{v}_{\mathsf{tot}}$ has been split, they may gain a considerable advantage in their attempts to learn $\mathsf{v}_{\mathsf{tot}}$.

We explore the circumstances under which k could be leaked.

The first and most evident risk is low network traffic, as it may lead to the k lock or burn transactions being submitted within a shorter delay with respect to other than other transactions on the network. We note that if there is enough traffic across Zclaim, this issue does not arise.

Furthermore, this risk can be mitigated by not submitting all sub-transactions simultaneously but within a short random delay between one another.

Furthermore, redeemers should use a different destination diversified payment address $(\mathsf{d}, \mathsf{pk}_{\mathsf{d}})$ per burn transaction, which may be derived from the same incoming viewing key. Otherwise, $\mathcal{A}$ may easily link burn transactions with each other in which vaults controlled by $\mathcal{A}$ are requested to redeem, hence allowing $\mathcal{A}$ to learn the note values.

Chapter 7

Conclusion and Outlook

In this work, we have shown that it is possible to preserve the privacy-preserving qualities of Zcash in cross-chain transfers. More generally speaking, we provide a specification for a decentralised cross-chain transfer protocol that fully integrates with a privacy-oriented cryptocurrency.

The author hopes that this can be of use to other teams attempting to interoperate with Zcash shielded transactions and that it encourages more privacy-friendly cross-chain transfer protocols to emerge.

7.1 Future Work

Listed below are a number of points on which improvements could or should be made, some of which may have been pointed out previously in this document:

- Concurrent Issue and Redeem procedures should be feasible with minor modifications.
- Block header verification must be adapted to the FlyClient upgrade [51], as well as inclusion proofs.
- The way issue and redeem availability is currently awarded is non-optimal and may reveal information about the transacted amount. Other approaches should be explored, such as the issuing chain 'handing out' issue and redeem requests to vaults at random, whereupon the vault chooses whether to accept it or not, instead of users picking a vault.
- A swap protocol as devised in XCLAIM may turn out non-trivial to design.

- It may prove beneficial to allow vaults to set transaction fees themselves, allowing them to fend off network congestion and at the same time incentivising competitiveness.
- The note encoding/challenging approach is interesting, but not very elegant. It may be possible after all to somehow enforce the correctness of the encoded components in a zk-SNARK instead.
- Finally, developing a working proof of concept would be the next step towards a full implementation.

Furthermore, Groth16, the zero-knowledge proving system employed in Sapling, is set to be replaced by the more efficient Halo2 [81] system, which also eliminates the need for a trusted setup, in a future upgrade. This will create a new liquidity pool, which in past updates has eventually resulted in older liquidity pools drying up, as pointed out by members of the Zcash development team. Hence Zclaim imperatively needs to be adapted to Halo2 before being implemented.

This work is planned to be summarised into a paper and submitted for publication in the coming months. Work on the project may be continued through a Zcash Open Major Grant [82].

Appendix A

Cryptographic schemes

Cryptographic schemes conceived for Zclaim are defined in the following pages. On the other hand, schemes taken from Sapling to which reference is made in this work were introduced in Chapter 3 and will not be reproduced here.

A.1 Commitment schemes

We define a commitment scheme as in [17, Section 4.1.7].

A.1.1 Nonce commitment scheme

The nonce commitment scheme NonceCommit may be instantiated as a Windowed Pedersen commitment scheme as defined in [17, Section 5.4.7.2] in a similar fashion to Sapling's note commitment scheme $\mathsf{NoteCommit}^{\mathsf{Sapling}}$, since the homomorphic properties required when hiding the note value are not necessary. It is defined as follows:

$$\mathsf{NonceCommit}_{\mathsf{rcn}}(\mathsf{n}_{\mathsf{permit}}) := \mathsf{WindowedPedersenCommit}_{\mathsf{rcn}}(\mathsf{n}_{\mathsf{permit}})$$

A.2 Signature schemes

We use the definition of a signature scheme in [17, Section 4.1.6].

A.2.1 Minting signature

MintingSig may be instantiated as RedJubjub as defined in [17, Section 5.4.6] without key re-randomisation and with generator

$$\mathcal{P}_{\mathrm{G}} = \mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(\text{“}\texttt{ZCLAIM\_Mint\_Sign}\text{”}, \text{“”})$$

A.2.2 Vault signature

VaultSig may be instantiated as RedJubjub as defined in [17, Section 5.4.6] without key re-randomisation and with generator

$\mathcal{P}_{\mathrm{G}} = \mathsf{DiversifyHash}(\mathsf{d})$

where d is the diversifier associated with the vault in the vault registry.

A.3 SIGHASH transaction hashing

We use the SIGHASH transaction hash as defined in [28], not associated with an input and using the SIGHASH type SIGHASH_ALL, to which we add two new fields:

- `hashShieldedMints` : $\mathbb{B}^{[256]}$ is 0 if the transaction does not contain a Mint transfer, otherwise it is the BLAKE2b-256 hash of the serialization of the Mint transfer (in its canonical transaction serialization format) with the personalisation field set to "`ZclaimMintHash`".
- `hashShieldedBurns` : $\mathbb{B}^{[256]}$ is 0 if the transaction does not contain a Burn transfer, otherwise it is the BLAKE2b-256 hash of the serialization of the Burn transfer (in its canonical transaction serialization format) with the personalisation field set to "`ZclaimBurnHash`".

Bibliography

[1] *CoinGecko: Cryptocurrency Prices & Market Capitalization*. URL: https://www.coingecko.com/ (visited on 29/11/2020).

[2] *GitHub - clearmatics/ion: General interoperability framework for trustless cross-system interaction*. URL: https://github.com/clearmatics/ion (visited on 19/12/2020).

[3] Markus Nissl et al. *Towards Cross-Blockchain Smart Contracts*. 2020. arXiv: 2010.07352 [cs.CR].

[4] Maurice Herlihy. 'Atomic Cross-Chain Swaps'. In: *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*. PODC '18. Egham, United Kingdom: Association for Computing Machinery, 2018, pp. 245–254. ISBN: 9781450357951. DOI: 10.1145/3212734.3212736.

[5] *Home · renproject/ren Wiki · GitHub*. URL: https://github.com/renproject/ren/wiki (visited on 19/12/2020).

[6] Alexei Zamyatin et al. *SoK: Communication Across Distributed Ledgers*. Working paper. Oct. 2019. URL: https://eprint.iacr.org/2019/1128.

[7] Will Warren and Amir Bandeali. *0x: An open protocol for decentralized exchange on the Ethereum blockchain*. White paper. 0x Project, 21st Feb. 2017. URL: https://0x.org/pdfs/0x_white_paper.pdf.

[8] Hayden Adams, Noah Zinsmeister and Dan Robinson. *Uniswap v2 Core*. White paper. Mar. 2020. URL: https://uniswap.org/whitepaper.pdf.

[9] Jae Kwon and Ethan Buchman. *Cosmos: A Network of Distributed Ledgers*. White paper. Tendermint Inc. URL: https://cosmos.network/cosmos-whitepaper.pdf.

[10] *GitHub - ZcashFoundation/zcash-pegzone: A shielded pegzone bridging Cosmos and Zcash.* URL: https://github.com/ZcashFoundation/zcash-pegzone (visited on 19/12/2020).

[11] Tokensoft Inc. *Wrapped*. URL: https://www.wrapped.com/wzec.html (visited on 22/12/2020).

[12] Joël Gugger. *Bitcoin-Monero Cross-chain Atomic Swap*. Cryptology ePrint Archive, Report 2020/1126. 2020. URL: https://eprint.iacr.org/2020/1126.

[13] *CCS - Monero Atomic Swaps implementation funding*. Sept. 2020. URL: https://ccs.getmonero.org/proposals/h4sh3d-atomic-swap-implementation.html (visited on 22/12/2020).

[14] Alexei Zamyatin et al. 'XCLAIM: Trustless, Interoperable, Cryptocurrency-Backed Assets'. In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. Mar. 2019, pp. 193–210. DOI: 10.1109/SP.2019.00085.

[15] Eli Ben-Sasson et al. 'Zerocash: Decentralized Anonymous Payments from Bitcoin'. In: *2014 IEEE Symposium on Security and Privacy*. IEEE. May 2014, pp. 459–474. DOI: 10.1109/SP.2014.36.

[16] Eli Ben-Sasson et al. *Zerocash: Decentralized Anonymous Payments from Bitcoin (extended version)*. Cryptology ePrint Archive: Report 2014/349. May 2014. URL: https://eprint.iacr.org/2014/349.

[17] Daira Hopwood et al. *Zcash Protocol Specification*. Technical specification. Version 2020.1.15 [Overwinter+Sapling]. Electric Coin Company, 6th Nov. 2020. URL: https://zips.z.cash/protocol/sapling.pdf.

[18] Adam Back et al. *Enabling Blockchain Innovations with Pegged Sidechains*. Tech. rep. 2014. URL: https://blockchainlab.com/pdf/sidechains.pdf.

[19] Vitalik Buterin. *Chain Interoperability*. Tech. rep. 9th Sept. 2016. URL: https://www.r3.com/wp-content/uploads/2017/06/chain_interoperability_r3.pdf.

[20] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. Bitcoin white paper. 2008. URL: https://bitcoin.org/bitcoin.pdf.

[21] Gavin Wood et al. *Ethereum: A secure decentralised generalised transaction ledger*. Ethereum yellow paper. Version 3e2c089. 5th Sept. 2020. URL: https://ethereum.github.io/yellowpaper/paper.pdf.

[22] *Ethereum Proof of Stake - EthHub*. URL: https://docs.ethhub.io/ethereum-roadmap/ethereum-2.0/proof-of-stake/ (visited on 17/11/2020).

[23] Ethan Buchman. 'Tendermint: Byzantine Fault Tolerance in the Age of Blockchains'. MA thesis. 2016. URL: http://hdl.handle.net/10214/9769.

[24] Aggelos Kiayias et al. 'Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol'. In: *Advances in Cryptology – CRYPTO 2017*. Ed. by Jonathan Katz and Hovav Shacham. Cham: Springer International Publishing, 2017, pp. 357–388. ISBN: 978-3-319-63688-7. DOI: 10.1007/978-3-319-63688-7_12.

[25] LM Goodman. *Tezos: a self-amending crypto-ledger*. Tech. rep. 2014. URL: https://tezos.com/static/white_paper-2dc8c02267a8fb86bd67a108199441bf.pdf.

[26] George Kappos et al. 'An Empirical Analysis of Anonymity in Zcash'. In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 463–477. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/kappos.

[27] *FIPS 180-4: Secure Hash Standard (SHS)*. NIST, Aug. 2015. DOI: NIST.FIPS.180-4.

[28] Jack Grigg and Daira Hopwood. *Transaction Signature Validation for Sapling*. Zcash Improvement Proposal 243. 10th Apr. 2018. URL: https://zips.z.cash/zip-0243 (visited on 16/12/2020).

[29] *Why is my transaction "pending"? — Coinbase Help*. URL: https://help.coinbase.com/en/coinbase/trading-and-funding/sending-or-receiving-cryptocurrency/why-is-my-transaction-pending (visited on 13/12/2020).

[30] *Cryptocurrency deposit processing times – Kraken*. URL: https://support.kraken.com/hc/en-us/articles/203325283-Cryptocurrency-deposit-processing-times (visited on 13/12/2020).

[31] *How long until my digital asset deposit reaches my account? – Support — Gemini*. URL: https://support.gemini.com/hc/en-us/articles/205424836-How-long-until-my-digital-asset-deposit-reaches-my-account (visited on 13/12/2020).

[32] *Blossom - Zcash*. URL: https://z.cash/upgrade/blossom/ (visited on 13/12/2020).

[33] *Wrapped ZEC - General - Zcash Community Forum*. URL: https://forum.zcashcommunity.com/t/wrapped-zec/35331 (visited on 25/11/2020).

[34] *BLAKE2b 'F' Compression Function Precompile · Issue #152 · ethereum/EIPs · GitHub*. URL: https://github.com/ethereum/EIPs/issues/152 (visited on 25/11/2020).

[35] *GitHub - ConsenSys/Project-Alchemy: Ethereum-Zcash Integration effort*. URL: https://github.com/ConsenSys/Project-Alchemy (visited on 05/12/2020).

[36] Markku-Juhani Saarinen and Jean-Philippe Aumasson. *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC): IETF RFC 7693*. Request for Comments 7693. Internet Engineering Task Force, Nov. 2015. DOI: 10.17487/RFC7693.

[37] Alex Biryukov and Dmitry Khovratovich. 'Equihash: Asymmetric Proof-of-Work Based on the Generalized Birthday Problem'. In: *Ledger* 2 (Apr. 2017), pp. 1–30. DOI: 10.5195/ledger.2017.48.

[38] Sean Bowe, Ariel Gabizon and Ian Miers. *Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model*. Cryptology ePrint Archive, Report 2017/1050. 2017. URL: https://eprint.iacr.org/2017/1050.

[39] *pairing/src/bls12_381 at e72660056e00c93d6b054dfb08ff34a1c67cb799 · zkcrypto/pairing · GitHub*. URL: https://github.com/zkcrypto/pairing/tree/e72660056e00c93d6b054dfb08ff34a1c67cb799/src/bls12_381 (visited on 09/12/2020).

[40] Jean-Philippe Aumasson et al. *BLAKE2 – fast secure hashing*. 22nd Feb. 2017. URL: https://www.blake2.net/#sp (visited on 26/12/2020).

[41] *Sapling integration in Tezos - Tech Preview*. 24th Dec. 2019. URL: https://blog.nomadic-labs.com/sapling-integration-in-tezos-tech-preview.html (visited on 25/11/2020).

[42] Antoine Rondelet and Michal Zajac. *ZETH: On Integrating Zerocash on Ethereum*. 2019. arXiv: 1904.00905 [cs.CR].

[43] *ZSL · ConsenSys/quorum Wiki · GitHub*. URL: https://github.com/ConsenSys/quorum/wiki/ZSL (visited on 27/11/2020).

[44] Zachary J. Williamson. *The AZTEC Protocol*. Dec. 2018. URL: https://github.com/AztecProtocol/AZTEC/raw/master/AZTEC.pdf.

[45] Xing Li et al. *Phantom: An Efficient Privacy Protocol Using zk-SNARKs Based on Smart Contracts*. Cryptology ePrint Archive, Report 2020/156. 2020. URL: https://eprint.iacr.org/2020/156.

[46] *GitHub - ZcashFoundation/zebra: An ongoing Rust implementation of a Zcash node*. URL: https://github.com/ZcashFoundation/zebra (visited on 27/11/2020).

[47] *GitHub - paritytech/parity-zcash: Rust implementation of Zcash protocol*. URL: https://github.com/paritytech/parity-zcash (visited on 27/11/2020).

[48] *GitHub - zcash/librustzcash: Rust-language assets for Zcash*. URL: https://github.com/zcash/librustzcash (visited on 27/11/2020).

[49] Yoav Nir and Adam Langley. *ChaCha20 and Poly1305 for IETF Protocols*. Request for Comments 7539. Internet Engineering Task Force, May 2015. DOI: 10.17487/RFC7539.

[50] *Plan how to change the proof-of-work · Issue #1211 · zcash/zcash · GitHub*. URL: https://github.com/zcash/zcash/issues/1211 (visited on 27/11/2020).

[51] Ying Tong Lai, James Prestwich and Georgios Konstantopoulos. *FlyClient - Consensus-Layer Changes*. Zcash Improvement Proposal 221. 30th Mar. 2019. URL: https://zips.z.cash/zip-0221 (visited on 27/11/2020).

[52] B. Bünz et al. 'FlyClient: Super-Light Clients for Cryptocurrencies'. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 928–946. DOI: 10.1109/SP40000.2020.00049.

[53] *Simplified Payment Verification (SPV) – BitcoinWiki*. URL: https://en.bitcoinwiki.org/wiki/Simplified_Payment_Verification (visited on 05/12/2020).

[54] *GitHub - ethereum/btcrelay: Ethereum contract for Bitcoin SPV: Live on https://etherscan.io/address/0x41f27...* URL: https://github.com/ethereum/btcrelay (visited on 02/12/2020).

[55] Dominik Harz et al. 'Balance: Dynamic Adjustment of Cryptocurrency Deposits'. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 1485–1502. ISBN: 9781450367479. DOI: 10.1145/3319535.3354221.

[56] *RenBridge FAQ - Darknodes*. URL: https://docs.renproject.io/darknodes/faq/renbridge-faq (visited on 09/12/2020).

[57] *Wrapped Bitcoin (WBTC) - CoinList*. URL: https://coinlist.co/asset/wrapped-bitcoin (visited on 09/12/2020).

[58] *Heartwood - Zcash*. URL: https://z.cash/upgrade/heartwood/ (visited on 19/12/2020).

[59] *Sprout-to-Sapling Migration — Zcash Documentation 4.1.0 documentation*. URL: https://zcash.readthedocs.io/en/latest/rtd_pages/sapling_turnstile.html (visited on 12/11/2020).

[60] Daira Hopwood and Eirik Ogilvie-Wigley. *Sprout to Sapling Migration*. Zcash Improvement Proposal 308. 27th Nov. 2018. URL: https://zips.z.cash/zip-0308 (visited on 16/12/2020).

[61] Ari Juels et al. *Mixicles: Simple Private Decentralized Finance*. Tech. rep. SmartContract Chainlink Ltd., 3rd Sept. 2019. URL: https://chain.link/mixicles.pdf.

[62] Salim Lahmiri, Stelios Bekiros and Antonio Salvi. 'Long-range memory, distributional variation and randomness of bitcoin volatility'. In: *Chaos Solitons & Fractals* 107 (2018), pp. 43–48. ISSN: 0960-0779. DOI: 10.1016/j.chaos.2017.12.018.

[63] Guglielmo Maria Caporale and Timur Zekokh. 'Modelling volatility of cryptocurrencies using Markov-Switching GARCH models'. In: *Research in International Business and Finance* 48 (2019), pp. 143–155. ISSN: 0275-5319. DOI: 10.1016/j.ribaf.2018.12.009.

[64] Jeffrey Quesnelle. *On the linkability of Zcash transactions*. 2017. arXiv: 1712.01210 [cs.CR].

[65] Ittay Eyal and Emin Gün Sirer. 'Majority is Not Enough: Bitcoin Mining is Vulnerable'. In: *Communications of the ACM* 61.7 (June 2018), pp. 95–102. ISSN: 0001-0782. DOI: 10.1145/3212998.

[66] Maria Apostolaki, Aviv Zohar and Laurent Vanbever. 'Hijacking Bitcoin: Routing Attacks on Cryptocurrencies'. In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. May 2017, pp. 375–392. DOI: 10.1109/SP.2017.29.

[67] Ethan Heilman et al. 'Eclipse Attacks on Bitcoin's Peer-to-Peer Network'. In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 129–144. ISBN: 978-1-939133-11-3. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/heilman.

[68] Karl Wüst and Arthur Gervais. *Ethereum eclipse attacks*. Tech. rep. ETH Zurich, 2016. DOI: 10.3929/ethz-a-010724205.

[69] Guangquan Xu et al. 'Am I Eclipsed? A Smart Detector of Eclipse Attacks for Ethereum'. In: *Computers & Security* 88 (Sept. 2019), p. 101604. DOI: 10.1016/j.cose.2019.101604.

[70] Bithin Alangot et al. *Decentralized Lightweight Detection of Eclipse Attacks on Bitcoin Clients*. July 2020. arXiv: 2007.02287 [cs.CR].

[71] Jack Peterson et al. *Augur: a Decentralized Oracle and Prediction Market Platform*. Feb. 2018. arXiv: 1501.01042 [cs.CR].

[72] John Adler et al. 'Astraea: A Decentralized Blockchain Oracle'. In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)* (July 2018), pp. 1145–1152. DOI: 10.1109/Cybermatics_2018.2018.00207.

[73] Steve Ellis, Ari Juels and Sergey Nazarov. *ChainLink: A Decentralized Oracle Network*. White paper. Version 1.0. SmartContract ChainLink Ltd., 4th Sept. 2017. URL: https://link.smartcontract.com/whitepaper.

[74] Sin Kuang Lo et al. 'Reliability analysis for blockchain oracles'. In: *Computers & Electrical Engineering* 83 (Feb. 2020), p. 106582. ISSN: 0045-7906. DOI: 10.1016/j.compeleceng.2020.106582.

[75] Imtiaz Mohammad Sifat and Azhar Mohamad. 'Circuit breakers as market stability levers: A survey of research, praxis, and challenges'. In: *International Journal of Finance & Economics* 24.3 (2019), pp. 1130–1169. DOI: 10.1002/ijfe.1709.

[76] John Fry and Jeremy Eng-Tuck Cheah. 'Negative bubbles and shocks in cryptocurrency markets'. In: *International Review of Financial Analysis* 47 (Feb. 2016), pp. 343–352. DOI: 10.1016/j.irfa.2016.02.008.

[77] SophonEX. *Does Cryptocurrency Have More Black Swan Events than Other Assets? — by SophonEX — SophonEX — Medium*. 1st Mar. 2019. URL: https://medium.com/sophonexchange/does-cryptocurrency-have-more-black-swan-events-than-other-assets-c274ad26a5c2 (visited on 12/11/2020).

[78] Jeremy Eng-Tuck Cheah and John Fry. 'Speculative bubbles in Bitcoin markets? An empirical investigation into the fundamental value of Bitcoin'. In: *Economics Letters* 130 (Feb. 2015), pp. 32–36. DOI: 10.1016/j.econlet.2015.02.029.

[79] *Regulation of Cryptocurrency Around the World*. Legal report. The Law Library of Congress, Global Legal Research Center, June 2018. URL: https://www.loc.gov/law/help/cryptocurrency/cryptocurrency-world-survey.pdf (visited on 18/11/2020).

[80] Karen Yeung. 'Regulation by Blockchain: the Emerging Battle for Supremacy between the Code of Law and Code as Law'. In: *The Modern Law Review* 82.2 (Mar. 2019), pp. 207–239. DOI: 10.1111/1468-2230.12399.

[81] *ECC releases code for Halo 2 - Electric Coin Company*. URL: https://electriccoin.co/blog/ecc-releases-code-for-halo-2/ (visited on 27/12/2020).

[82] *About us - Zcash Open Major Grants (ZOMG)*. URL: https://zcashomg.org/ (visited on 27/12/2020).

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

CONFIDENTIAL CROSS-BLOCKCHAIN EXCHANGES: DESIGNING A PRIVACY-PRESERVING INTEROPERABILITY SCHEME

Authored by (in block letters):

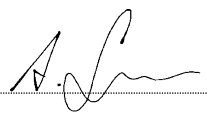
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Sanchez Keller	Aleixo

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 27/12/2020	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.