ZKFlow: Private Transactions in Corda with ZKP

Matthijs van den Bos Victor Ermolaev Scott King
Alexey Koren Gamze Tillem

ING Bank, The Netherlands

blockchain@ing.com

Abstract

Corda is a permissioned distributed ledger platform that enables enterprises to manage their business deals and obligations in a decentralized manner. Its unique ledger structure, which requires sharing the content of a transaction only to participating parties, contributes to the protection of privacy-sensitive transaction data. Despite improved privacy guarantees, the platform cannot assure complete data protection due to two main issues: the leakage of transaction data to the validating notaries and the leakage of historical transaction data in the backchain to future participants. These issues can be handled using cryptographic techniques whose adoption in Corda requires a dedicated protocol design.

In this paper, we present a protocol that overcomes the problem of data leakage in the notarization and transaction backchain validation of Corda transactions using zero-knowledge proofs (ZKPs). Our protocol enables the platform participants to validate their transactions without revealing their content to the non-trusted parties. We provide the technical details of our protocol and show how ZKPs can be efficiently adopted into the Corda ecosystem.

Contents

List of Figures

1 Introduction

Corda is a permissioned distributed ledger platform that enables enterprises to manage their business deals and obligations in a decentralized manner. The efficient and scalable structure of the platform made it suitable for different use cases in several industries such as healthcare, telecommunications, and capital markets [18, 7]. A unique property of Corda ledger, which requires to share the content of a transaction only with the parties involved in it, boosts the performance of the platform [7]. This property additionally contributes to data privacy since access to transaction content is restricted to a permitted group of users.

While the ledger structure of the Corda platform contributes to data privacy, it does not guarantee complete privacy protection on the platform. In the current design, there are mainly two privacy concerns:

Privacy of notarisation: The consensus mechanism in Corda comprises notary services, who provide uniqueness consensus and contribute to validity consensus as a required signer. The platform provides two types of notary services: validating notaries and non-validating notaries [11]. Validating notaries only approve and sign a transaction after validating the transaction content. To do that, a validating notary has full visibility of the transaction content and the entire transaction history back to the point of state issuance [13]. Granting full visibility of their data to a possible untrustworthy third party is not desirable for network participants due to privacy concerns associated with compliance and operational risks. In practice, this means the validating notary is seldom used in production settings. As of the time of publishing, the Corda Network notary is a non-validating notary [16].

Non-validating notaries overcome the privacy problem of Corda ledger by restricting access for notaries to only certain public information such as the consumed state identifiers: they do not validate the full transaction content, nor its history. Instead, non-validating notaries verify whether the state identifiers of the transaction have not been spent by checking their list of spent states. While restricting access to transaction content eliminates the privacy problem, it might lead to a denial-of-state attack (DoSt), which enables malicious actors to spend existing states that do not belong to them. We refer readers to [13] for the details of the DoSt attack.

Privacy of transaction backchain: Before sending a new transaction to the notary for validation, each participant of the transaction not only validates the current transaction contents, but also the contents of all historical

transaction leading up to it. Only then they will be convinced that the transaction is valid. To do so, the participants will resolve the transaction backchain for each input and reference state in the current transaction, up until their creation transaction. As a participant of the current transaction, it is legitimate to have access to the full contents of the current transaction. However, granting participants access to the transaction backchain content is not desirable for the participants that were involved in the historical transactions due to the leakage of their potentially sensitive data to future participants in the transaction chain.

Trivial solutions to overcome the privacy problems in Corda exist such as re-issuance of assets, keeping data off-ledger, or using non-validating notaries despite the risk of a DoSt attack. These workarounds do not necessarily achieve full privacy protection and might be still vulnerable to certain attacks [13]. Nevertheless, two techniques that have been investigated thoroughly for data protection in the last couple of years stand out as promising solutions for the privacy concerns in Corda: trusted hardware and zero-knowledge proofs (ZKPs).

Trusted hardware is a specialized hardware design that resists adversarial access with the usage of some cryptographic techniques [19]. Intel's Software Guard Extensions (SGX) is a well-known example of trusted hardware design, which allows users to create private regions within memory, i.e., enclaves that assure confidentiality and integrity in the presence of any malicious actor, including the owner of the hardware [15]. Despite its computational efficiency in the processing of sensitive data, the frequent vulnerabilities reported on the SGX architecture remains a drawback in the adoption of SGX by DLT systems [10, 11, 13]. Furthermore, requiring the vendor, i.e., Intel, as the trust anchor weakens the trust proposition of SGX [14].

Zero-knowledge proofs provide an alternative solution to the privacy problems in Corda by eliminating the issues around the vendor dependency and hardware vulnerabilities. ZKPs are cryptographic protocols that enable a party to prove their knowledge of certain information to another party without leaking anything about that information. While ZKPs have been proposed as a theoretical concept in the 1980s, current ZKP schemes achieve practical performance that can be used in modern digital systems [9]. ZKPs have several shortcomings compared to hardware solutions regarding their performance, limited flexibility, and dependency on non-standard cryptography. Due to the underlying advanced cryptographic primitives, ZKPs involve non-trivial computations, which limits their efficiency and flexibility. However, they are still practical to be used in current systems. Zcash is a prominent example for the real world application of ZKPs [12]. Furthermore, the extensive research effort around ZKPs can speed up their standardization

and help mitigate performance issues. Additionally, their provable security, which is based on strong mathematical assumptions, makes ZKPs a viable alternative for overcoming privacy challenges in Corda.

In this paper, we present a zero-knowledge transaction validation protocol for Corda, ZKFlow, that overcomes the privacy problem in Corda's transaction validation mechanism in the existence of potentially non-trusted participants. Our protocol uses zero-knowledge proofs to achieve data protection. Using ZKFlow, transaction participants can validate the backchain of a transaction that they are involved without having access to the historical sensitive state content. Similary, a notary can validate the contents of a transaction and its backchains without seeing any of the transaction contents. We explain how a Corda transaction can be converted to a zero-knowledge transaction while adhering to the original Corda communication flow. Our protocol provides practical proving and verifying time for Corda transaction validation such that validation time can be even shorter than the original one due to the constant verification time of ZKPs. As an example, for a transaction of size 12KB, ZKFlow can perform ZKP setup in 4 minutes, generate the proof in 2 minutes, verify it in 20 ms on a standard MacBook Pro.

Our protocol builds on top of Corda's original transaction protocol and extends it with privacy-preserving features. Therefore, in the explanation of the protocol we use the terms and concepts that are defined in the Corda technical paper and documentations [11]. We refer the readers who are not familiar with these concepts to the resources provided by the Corda platform.

Scope of the paper: This paper focuses on how to perform transaction validation in Corda using ZKPs: the transaction validation protocol and circuit management (i.e. setup, updates). Integrating ZKP into Corda requires handling several other tasks such as serializarion of transaction components and selection of ZKP scheme.

One aspect of integration of ZKPs into Corda is the serialization of transaction components. ZKFlow requires each transaction to be of a deterministic structure and fixed size. We addressed this with a custom serialization scheme and transaction structure metadata. Since both do not affect the transaction validation protocol design, they are out of the scope of this paper. We cover them in the design documentation of the ZKFlow protocol.

Another clarification for the scope of our paper is the selection criteria for the ZKP scheme. Our protocol does not target a specific ZKP scheme. Therefore, selection of ZKP scheme does not affect the design of our protocol. Depending on the use case and risk appetite, different schemes can be used such as with or without trusted setup, faster or slower proving and

verification, and smaller or larger key and proof sizes.

In the remainder of the paper, we first provide a brief overview of our ZKFlow protocol in Section 2. Then, we provide the preliminary information on the cryptographic primitives in Section 3, on the transaction structure in Section 4, and on the transaction backchain validation in Section 5. In Section 6, we explain the content of the ZKP used in ZKFlow. Based on the preliminary knowledge provided in the former sections, in Section 7, we explain the ZKFlow in detail. In Section 8, we explain how circuit setup is performed and the artefacts are distributed. In Section 9, we provide an example of transaction validation. We discuss several limitations of ZKPs and how they are addressed in our protocol in Section 10 and we conclude the paper in Section 11.

2 Protocol Overview

Transaction validation in Corda involves mainly three types of parties: 1. *an initiator*, who creates a new transaction, 2. *a counterparty*, who is the intended recipient of the new transaction, 3. *a notary*, who notarizes the transaction. In the original Corda protocol, once an initiator builds a valid Corda transaction, they should validate it via

- a counterparty that validates
  - the content of the transaction by recomputing its transaction id from its content that is **fully visible** to the counterparty,
  - the backchain of the transaction by resolving the historical transactions up until the issuance transaction, all of which are **fully visible** to the counterparty,
  - the smart contract rules of the current transaction.
- and a notary, where a validating notary validates
  - the content of the transaction by recomputing its transaction id from its **fully visible** content,
  - the backchain of the transaction, which is **fully visible** to the validating notary,
  - and the smart contract rules [11].

In ZKFlow, we change this validation procedure by restricting access of the counterparties and notaries to privacy sensitive data. Therefore, the validation of private data is only possible with zero-knowledge proofs. In the

protocol, the initiator is responsible of a valid transaction generation. The initiator, a.k.a., *prover*, should prove to the counterparty and the notary, a.k.a., *verifiers*, that the transaction is valid while preserving confidentiality. In the case of counterparty, the confidential information is the sensitive state data that is stored in the transaction backchain. In the case of notary, the content of current transaction and the transaction backchain is the confidential information. In the explanation of our protocol, the terms *prover* and *initiator*, and *verifier* and *notary/counterparty* are used interchangeably. To perform a transaction validation using ZKPs, the participants need proving and verifying keys. Therefore, a ZKP setup must be performed to generate these keys. When there is a change in the transaction structure or validation rules, the setup phase might be repeated if the underlying ZKP scheme does not support universal or updateable setup.

Figure 1 illustrates the subprotocols of ZKFlow with the interaction among the participating parties during the zero-knowledge transaction validation. The setup phase, where the cryptographic parameters for the ZKP are generated, is performed beforehand as a one time operation. Thus, it is not included in the protocol overview.

In ZKFlow, once an initiator builds a valid Corda transaction, it should perform three subprotocols:

- a **transaction signing and backchain validation protocol** with a counterparty, where a counterparty validates
  - the content of the transaction by recomputing its transaction id from its content that is **fully visible** to the counterparty,
  - the backchain of the transaction by resolving the historical transactions up until the issuance transaction, where the counterparty has **limited visibility** for each historical transaction that consists of a certain public information accompanied with a zero-knowledge proof of its private content,
  - the smart contract rules of the current transaction.
- a **transaction notarization protocol** with a notary, where the notary validates
  - the content of the transaction by recomputing its transaction id with **limited visibility** of the transaction content,
  - the historical transactions by validating the transaction backchain based on a certain public information and the accompanying ZKPs,
  - and the smart contract rules.

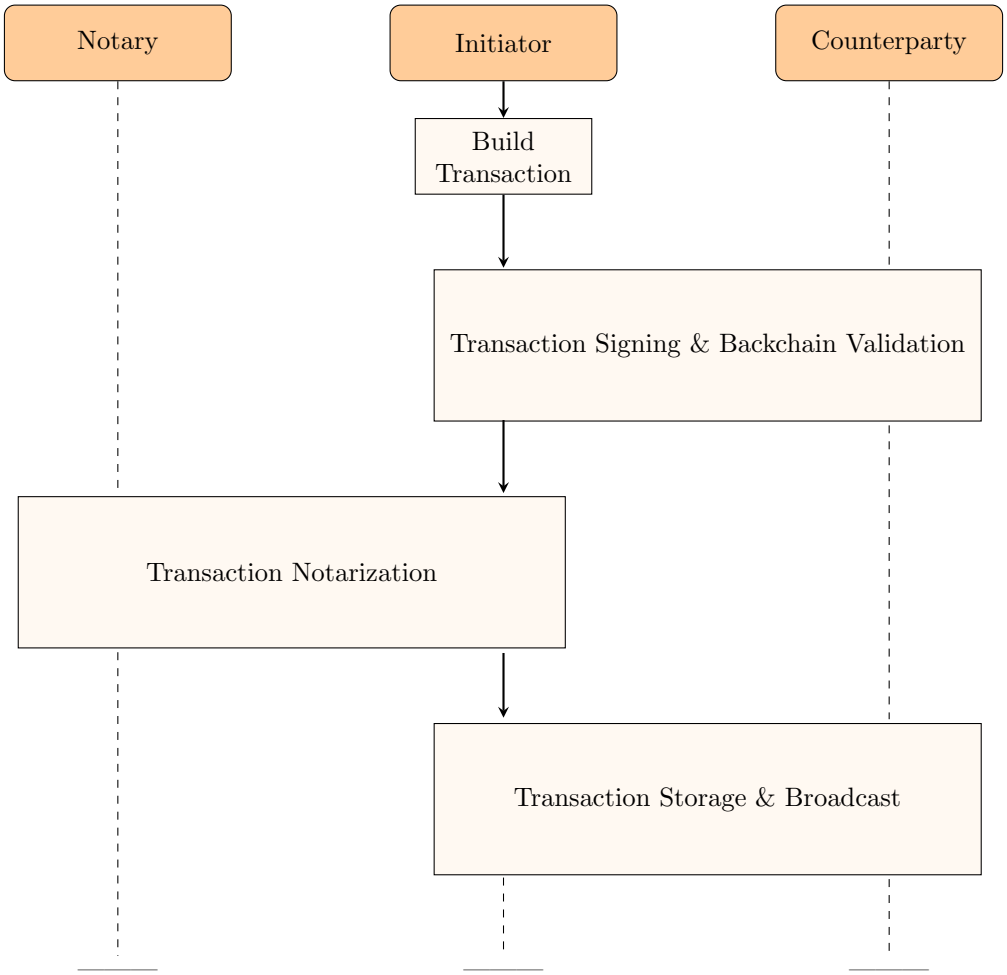

Figure 1: ZKFlow main protocol.

- and a **transaction storage and broadcasting protocol**, where the initiator stores the transaction along with its signatures and broadcasts the notary signature to the counterparty(s) for storage.

Before diving into the details of each subprotocol, it is useful to describe some of the building blocks of ZKFlow. Hence, in the following sections, we explain the transaction structure used in ZKFlow, how a party can validate the backchain of a transaction, and what is proven with a zero-knowledge proof. We also provide a brief summary on the cryptographic primitives used. We explain the details of the ZKFlow in Section 7.

3 Cryptographic Preliminaries

The ZKFlow protocol uses several cryptographic primitives to achieve efficient private transaction validation. In the following, we provide a brief explanation of the cryptographic primitives used in our design.

3.1 Zero-Knowledge Proofs

Zero-knowledge proofs enable a prover to prove to a verifier that they possess the knowledge of information that satisfies certain criteria without revealing the information itself. A ZKP scheme must satisfy three properties:

- **Completeness:** If the proof statement is true, an honest prover can convince an honest verifier.
- **Soundness:** If the proof statement is false, a cheating prover can convince an honest verifier only with some negligible probability.
- **Zero-knowledge:** The verifier can only learn the correctness of the statement and nothing else.

Since their introduction in 1980s by Goldwasser et al. [9], many improvements and alternative schemes are proposed for ZKPs that made them practical for modern digital systems. Today's practical ZKP solutions, e.g. zk-SNARKs [4],

- are non-interactive i.e., the prover and the verifier do not need to be online at the same time during proving
- can generate succinct proofs, i.e., the proof size and the verification time can be constant.

When proving a statement with a ZKP, a prover generates a proof on the secret information, i.e., *witness* w, and provides the proof along with some public information, i.e., **instance x**, to the verifier. The proof is performed in three steps:

- $\textsc{setup}(1^\lambda) \rightarrow (PK, VK)$: Non-interactive ZKPs require generation of certain information beforehand to perform proof and verification operations. In the setup phase, the proof circuit is constructed and the proving key PK and the verifying key VK are generated based on a security parameter 1^λ. The setup constitutes the majority of the computation cost in ZKPs. However, it is a one time operation and does not need to be repeated unless there is a change in the ZKP circuit structure.

- PROVE$(PK, \mathbf{x}, w) \rightarrow \pi$: A prover can generate a proof π on their private witness using the proving key PK. The prove phase is significantly shorter time than the setup phase. Some ZKP schemes allow fixed-size proof generation.

- VERIFY$(VK, \mathbf{x}, \pi) \rightarrow \mathit{bool}$: Once receiving a proof, any verifier that has access to the verification keys can verify a ZKP. In succinct ZKPs, verification can be performed efficiently in short time or in constant time [4].

3.2 Hash Functions

The transaction validation in Corda requires to compute certain hash functions on the transaction data. In our protocol, we need to perform these hash operations within the ZKP circuit since the input of the hash function might be privacy-sensitive. However, the hash functions used by the Corda platform, such as SHA256, cannot perform efficiently with ZKPs. These type of standard functions are based on nonlinear gates, which are expensive in arithmetic circuits that are used in the design of ZKPs. Therefore, in our protocol, we use Blake hash function in the transaction validation that can perform efficiently on ZKPs.

3.2.1 Blake2 Hash

Blake2 is an improvement of Blake hash function that was a candidate for SHA3 [2]. In our protocol, we use Blake2s variant which has 64 bytes input size and 32 bytes digest size. An important property of Blake2 is that it is resistant to length extension attack, which means our protocol does not require the double hashing operations as in the original Corda protocol. Blake2 is more expensive in ZKP circuits than other hash functions that are designed for arithmetic circuits. However, it is more efficient compared to SHA256. In the rest of the paper, we use the notation $\mathbf{BH}(x) = \mathbf{Blake2s}(x)$, where $\mathbf{Blake2s}(x)$ represents the hash computation operation on the preimage x and $\mathbf{BH}(x)$ represents the hash digest.

4 Transaction Structure

In our protocol, the computation of the transaction id is same as the original Corda. However, the structure of transaction is moderately limited compared to original Corda transactions considering the privacy protection and

efficiency of computations. In ZKFlow, to prevent leakage of privacy sensitive data, we use two different transaction types, which are

- a private *WireTransaction* that contains the original transaction content and can only be accessed by the owners of transaction, i.e., the initiator and the counterparty, and
- a public *ZKVerifierTransaction* that contains a filtered transaction[1] and the ZKP that is computed on the witness WireTransaction, and is accessible by the notary and the future counterparties.

In this section, we explain the structure of both transaction types and how the transaction id is computed for each transaction type.

4.1 The structure of a Corda WireTransaction

To aid understanding of the structure of a ZKVerifierTransaction, we first describe the structure of a standard Corda WireTransaction. On a high level, a transaction in Corda consists of several components such as inputs, outputs, references, commands, signers, attachments, time window, notary, network parameters. A transaction should contain at least one command, one signer that is associated with the transaction, and an input or output. The rest of the components, such as attachments, are optional and not necessarily included in the transaction.

In the case of ZKFlow, the cardinality of each component and the content of components are crucial since their size affect the computations in the ZKP circuit. More explicitly, the number of items in each component group should be fixed to a predetermined number. Changing the number of components results in a new ZKP circuit which indicates that the setup phase should be repeated for each transaction, which is not desirable and practical. Furthermore, the size of each component element should be fixed. Therefore, arbitrary size elements are not allowed in our protocol since they require a new setup and also larger input sizes affect the performance of computations significantly.

The following are the components that are included in a ZKFlow WireTransaction:

- **inputs**, `StateRef` objects that points to a UTXO meant to be consumed,

[1] `https://docs.corda.net/docs/corda-os/4.7/tutorial-tear-offs.html#introduction`

- **references**, `StateRef` objects that points to a UTXO meant to be referenced and not consumed,
- **outputs**, that are `TransactionState` objects,
- **commandData**, the command associated with the transaction, such as Issue, Move, Redeem, etc.,
- **commandSigners**, the public keys of the parties that sign the transaction,
- **notary**, the service that signs the transaction,
- **attachments**, the SHA256 hashes (AttachmentId) of any file or document that is related to the transaction,
- **timeWindow**, the period that the transaction should be validated within,
- **networkParametersHash**, the hash of the set of parameters that are used for correct interoperation among participant nodes in a network.

4.1.1 Calculation of Transaction id

The transaction id identifies a WireTransaction. It is deterministically calculated from the contents of all transaction components. Thus, if any part of the transaction changes, the transaction id will change. This can be used to prove the integrity of the transaction: that transaction components actually belong to the transaction. For a valid transaction id, it should be possible to do the following:

- Calculate the transaction id efficiently in an arithmetic circuit context, given a WireTransaction,
- Calculate the transaction id, given a ZKVerifierTransaction,
- Efficiently prove the membership of one component in a transaction with a certain transaction id.

Corda uses a Merkle tree to represent the transaction data structure and calculate the transaction id based on its contents. In Figure 2, we provide an illustration of WireTransaction that is represented with a Merkle tree structure. Elements of component groups are stored in the leaf nodes of the tree. In the figure, we use different colors to differentiate the representation of inputs, outputs, and references from other component groups. This

representation will be useful later in the validation of transaction backchain (Section 5).

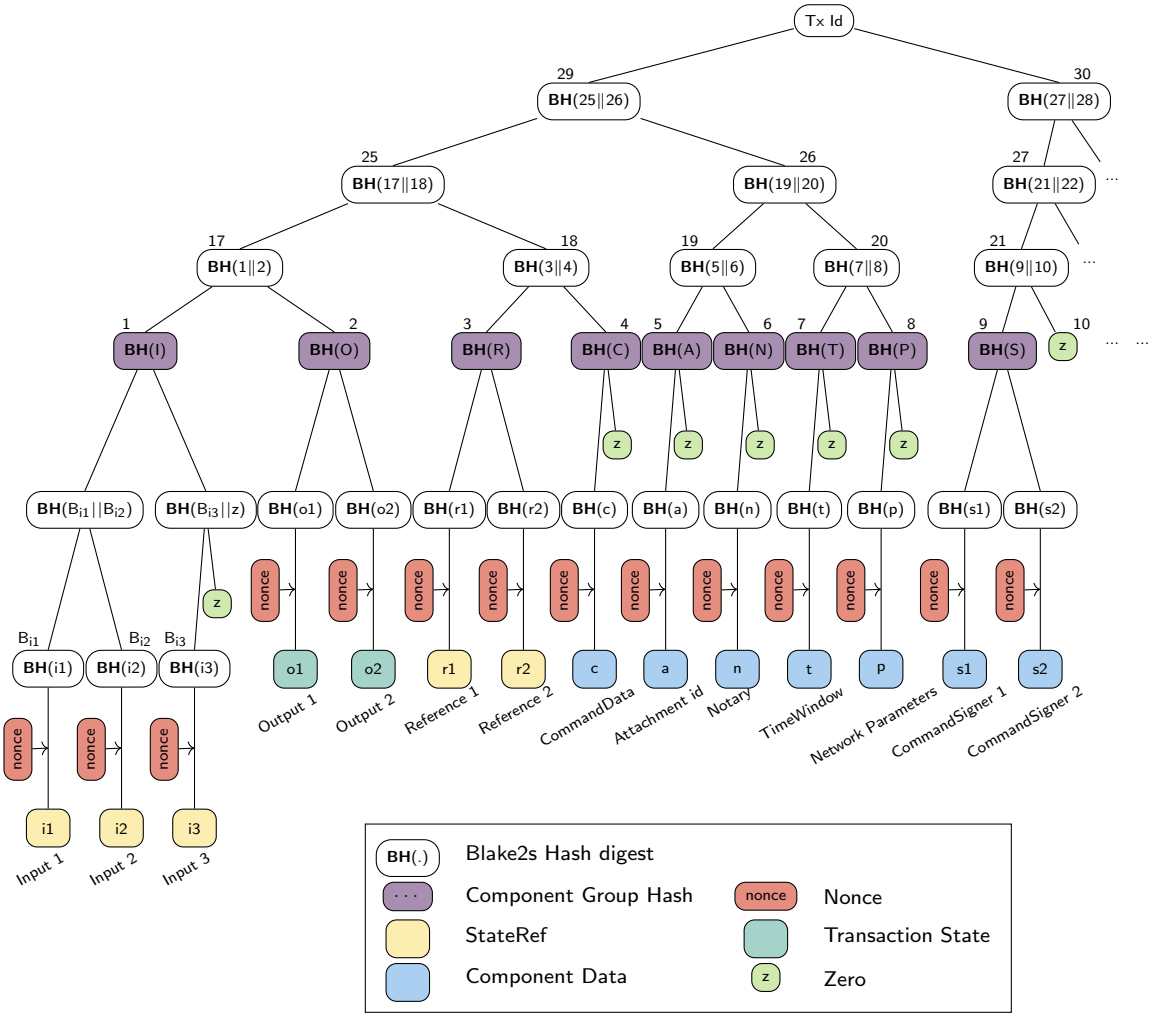

Figure 2: Merkle Tree representation of a transaction.

To represent a transaction with a Merkle tree in ZKFlow, we use Blake2s hash function in the computation of nonces, component leaf hashes and internal nodes . This is different from Corda, which uses double SHA-256 for performance and security reasons. Our choice of hash function does not affect the structure of the transaction and Merkle tree.

The computations on the Merkle tree start from the leaf nodes. Each component group corresponds to a Merkle subtree, where the root gives the component group hash (represented with purple nodes in Figure 2). After the

Merkle root of each subtree is computed, the component group hashes used to calculate the Merkle root of main tree, which is equal to the transaction id. In Corda's Merkle tree computation algorithm, when there is a single element in a component group, an auxiliary input (zeros) is added to the group to make its size a positive integer power of two. Then, the hash of these two elements become the component group hash. When the number of leaves of a Merkle tree is greater than one and not an integer power of two, then the auxiliary elements are added to the tree until the number of leaves is an integer power of two. In Figure 2, the green boxes represent these padded elements. On the component group hashes level, seven of the auxiliary zero values are added to the tree to assure a Merkle tree of size $2^4 = 16$.

Calculating the hash of a transaction component In the computation of the transaction id, the first step is to compute the leaf hash of each component element. To avoid collisions and pre-image attacks, each leaf node is prefixed with a nonce that is unique for each element within the respective component group.

Given a 256-bits uniform privacy salt $s \in_R \{0,1\}^{256}$, and a transaction $Tx = (C_1, \ldots, C_n)$ consisting of an ordered list of n components groups C, such that each component group $C_i = (c_{i,1}, \ldots, c_{i,m})$ consists of an ordered list of m components $c_{i,j}$, the nonce $n_{i,j}$ for a transaction component $c_{i,j}$ in the component group C_i is

$$n_{i,j} = \textbf{Blake2s}(s \,\|\, i \,\|\, j), \text{ s.t. } \; i \in \{1, n\} \text{ and } j \in \{1, m\}.$$

Using this deterministic nonce, the leaf hash of transaction component $c_{i,j}$ in component group C_i is computed as

$$h_{i,j} = \textbf{Blake2s}(n_{i,j} \,\|\, c_{i,j}).$$

In the computation of leaf hashes, the component elements $c_{i,j}$ should be serialized to be used as an input to the hash function such that their serialization can deterministically and uniquely represent them.

Computing the root of the Merkle tree In the computation of the Merkle root, the first step is to compute the component hashes. In the Merkle tree representation, each component group C_i is represented by a subtree such that the components are the leaf nodes of the subtree. Initially, the hash of each element within C_i is computed using Blake2s such that

$$h^0_{i,j} = \textbf{Blake2s}(n_{i,j} \,\|\, c_{i,j}),$$

where the superscript 0 in $h^0_{i,j}$ represents the corresponding level for the hash in the subtree and $n_{i,j}$ is the nonce that is generated as explained above using Blake2s hash function. If the number of elements in C_i is m, then the depth of the subtree that represents C_i becomes $\Delta = \lceil \log_2 m \rceil$. The list of the hash values in the leaf level of a subtree is represented as $H^0_i = (h^0_{i,1}, \ldots, h^0_{i,m})$. To compute the hashes on a level d of the subtree, the child nodes are concatenated and hashed with Blake2s hash such that

$$h^d_{i,j} = \textbf{Blake2s}(h^{d-1}_{i,2j} \parallel h^{d-1}_{i,2j+1}).$$

Then, the list of hash values in level d is represented as

$$H^d_i = \left(h^d_{i,1}, \ldots, h^d_{i,\lfloor \frac{m}{2^d} \rfloor} \right).$$

Finally, the component group hash of the component group C_i is a single value such that

$$H_i = (h^{\Delta}_{i,\lfloor \frac{m}{2^{\Delta}} \rfloor}), \text{ where } \left\lfloor \frac{m}{2^{\Delta}} \right\rfloor = 0 \text{ since } \Delta = \lceil \log_2 m \rceil.$$

In the computation of internal hash values, if the number of the nodes in a level of a component subtree is not multiple of two, then we use an auxiliary input, zero (represented with green boxes in Figure 2), which is formed as 32 bytes of 0s. In that case, the Blake2s hash is computed as follows:

$$h^d_{i,j} = \textbf{Blake2s}(h^{d-1}_{i,2j} || zero).$$

Once the component group hashes are computed, the computations in the main Merkle tree are the same as the subtrees. Given that $\mathcal{H}^0 = \{\mathcal{H}^0_0, \cdots \mathcal{H}^0_t\}$ is the set of component group hashes (which are represented as H_i above), the following operation is repeated in each level of the main Merkle tree to compute the transaction id:

$$\mathcal{H}^{\omega}_i = \textbf{Blake2s}(\mathcal{H}^{\omega-1}_{2i} \parallel \mathcal{H}^{\omega-1}_{2i+1}), \text{ such that } \mathcal{H}^{\omega} = \{\mathcal{H}^{\omega}_0, \cdots \mathcal{H}^{\omega}_{\lfloor \frac{t}{2^{\omega}} \rfloor}\},$$

where ω is the current depth in the main Merkle tree, $\mathcal{H}^{\omega}$ is the set of hashes on level ω, and $\lfloor \frac{t}{2^{\omega}} \rfloor$ is the number of elements in $\mathcal{H}^{\omega}$. When $\lfloor \frac{t}{2^{\omega}} \rfloor$ is not even, the hash operation must be performed using zero-padding as $\mathcal{H}^{\omega}_i = \textbf{Blake2s}(\mathcal{H}^{\omega-1}_{2i} \parallel zero)$, where zero is formed as 32 bytes of 0s. Given that the depth of the main tree is Ω, the Merkle root is

$$\text{Transaction Id} = \mathcal{H}^{\Omega}_0 = \textbf{Blake2s}(\mathcal{H}^{\Omega-1}_0 \parallel \mathcal{H}^{\Omega-1}_1).$$

4.2 The structure of a ZKVerifierTransaction

Different from the WireTransaction, a ZKVerifierTransaction contains much less information regarding the transaction content to prevent leakage of sensitive data, most importantly the output contents, to the verifier. The ZKVerifierTransaction is a filtered transaction with the zero-knowledge proof that validates the WireTransaction. The transaction id of the ZKVerifierTransaction is identical to the WireTransaction. Sharing less information with the verifier does not threaten the validity of a transaction and does not give an advantage to the prover to cheat in validation since the ZKP and the available public information is sufficient to validate the transaction. The information shared by the verifier in a ZKVerifierTransaction is determined by the visibility of the corresponding component. In ZKFlow, we use three types of visibility for each component:

- **Private**, the content of the component is not visible. The verifier can validate the component only with the help of a ZKP.
- **Public**, the content of the component is visible. The verifier can publicly validate the component.
- **Mixed**, the content of the component is visible. It can be used in the ZKP to help validating other private components.

In ZKFlow, the default visibility rules of components in a ZKVerifierTransaction are:

Public:

- **inputs**, `StateRef` objects that points to a UTXO meant to be consumed,
- **references**, `StateRef` objects that points to a UTXO meant to be referenced and not consumed,
- **commandData**, the command associated with the transaction,
- **notary**, the service that signs the transaction,
- **timeWindow**, the period that the transaction should be validated within,
- **networkParametersHash**, the hash of the set of parameters that are used for correct interoperation among participant nodes in a network.

- **commandSigners**, the public keys of the parties that should sign the transaction.

Public, represented by their group hash:

- **attachments** group hash, the component group hash of attachments.

Private, only leaf hashes visible:

- **outputs**, the leaf hashes of the output states of the WireTransaction that are digests of Pedersen hash function,

Depending on the contract validation rules and privacy concerns, the visibility of components can be changed. Visibility rules are specific to each element in a component group. Thus, a component group can have private, public and mixed components at the same time.

In Figure 3, we illustrate the structure of the ZKVerifierTransaction with default component visibilities. The data stored in the black nodes in the figure is **not** shared with the verifier due to their privacy sensitive content regarding the default visibility rules of ZKFlow. The data shared with the verifier is required for the validation of the transaction and does not leak information. The information leaked to the verifier is the the command used in the transaction, the number of outputs, and the public keys of the signers.

The command is leaked to identify the circuit type that is used for the ZKP. This leakage is due to performance/privacy trade-off. Since each command has different input-output structure and contract rules, using a circuit that covers all commands might result in a significant performance bottleneck. Thus, to optimize the cost of computations, we create a separate ZKP circuit for each command that is associated with components with private and/or mixed visibility, which is called a `ZKCommand`. Creating the proof per `ZKCommand` still protects the sensitive transaction content but only leaks the command type of the transaction. The number of outputs is also leaked intentionally for the validation of the backchain. The leaf hashes that corresponds to the output states are used in subsequent transactions for backchain validation as public information. We provide more explanation on this in Section 5.

Other information that is leaked due to performance/privacy trade-off is the number of signers and the public keys that are associated with the signers. Including the signature validation within the ZKP circuit increases the cost of computations in the setup and proving times. Furthermore, similar to hash functions, the signature schemes and their underlying elliptic curves that are efficient in arithmetic circuits are not supported in the current Corda design. We decided to use the signature schemes available in the platform

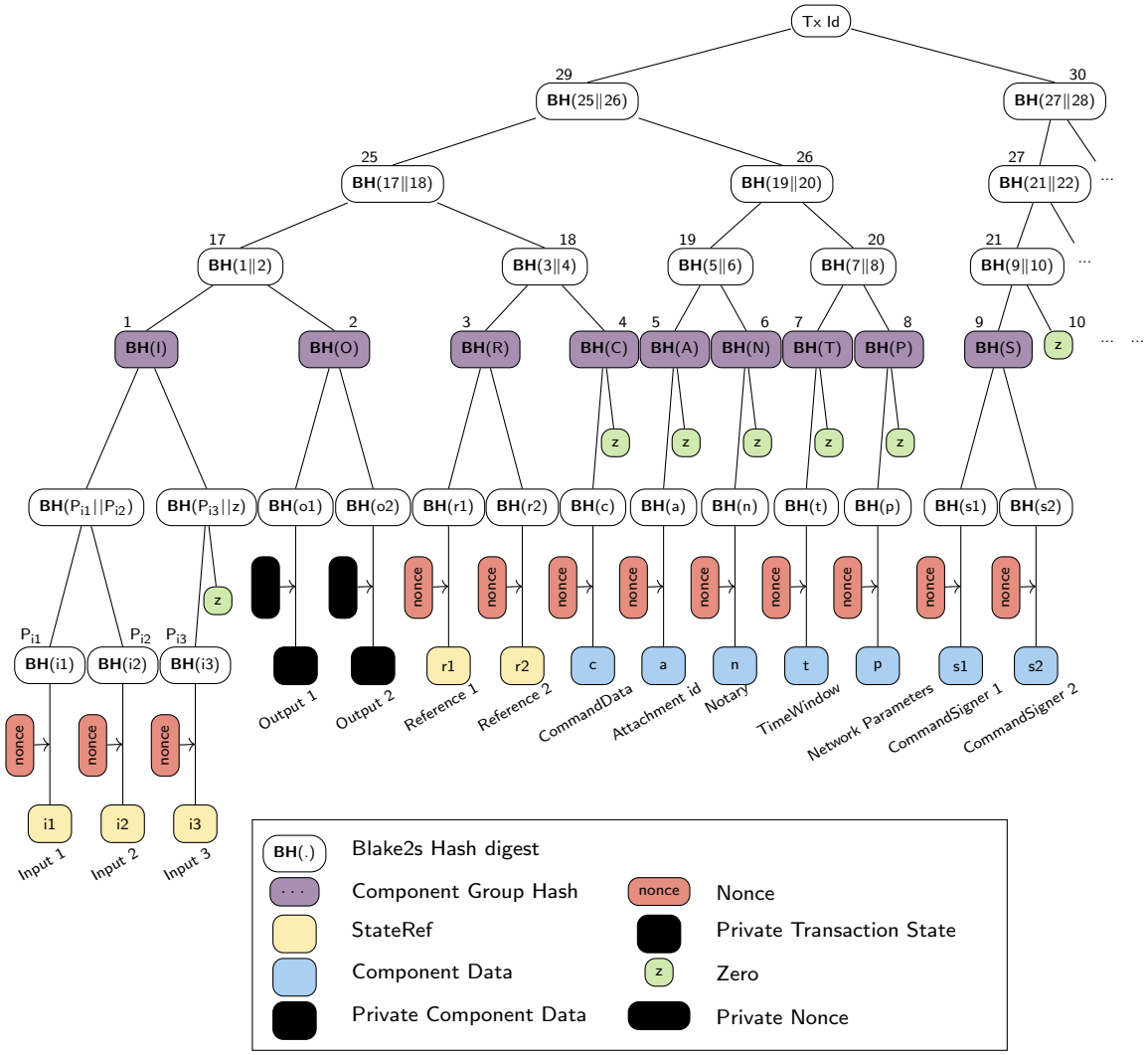

Figure 3: Merkle Tree representation of a ZKVerifierTransaction.

and move the validation of signatures outside of the circuit. If the identity of transacting parties is considered sensitive information in combination with the visible components of a ZKVerifierTransaction, the confidential identities offered by Corda can help assuring the anonymity of signers [8].

In the computation of ZKVerifierTransaction, the verifier cannot validate the nonce values for public components since they do not have access to the privacy salt. Granting verifier access to the privacy salt threatens the sensitivity of private components. A curious verifier can compute the nonces for private components using privacy salt and might reveal the value of the

component from component leaf hash with some external knowledge on the component. On the other hand, a corrupted initiator might send a fake nonce and component pair in ZKVerifierTransaction.

The possibility that a malicious initiator can perform such an attack is negligible given the collision-resistance guarantee of the underlying hash function. In ZKFlow, we use Blake2s to compute component leaf hashes. Thus, using Blake2s , a malicious initiator can find any two nonce-component pairs in 2^{128} attempts, which means the attacker should also alter the corresponding WireTransaction. If the attacker cannot modify the corresponding WireTransaction, then the effort that an attacker spends to find a nonce-component pair that has the same component leaf hash value as the original one requires 2^{256} attempts. Given that 128-bits security is sufficient to assure security with current standards [2] and even lower levels of security (down to 100-bits) is acceptable with current computational power [1], we do not see any threat to the validity of ZKFlow.

4.2.1 Calculation of Transaction Id

In the computation of transaction id from a ZKVerifierTransaction, the main difference from a WireTransaction is the content of each component group. In the ZKVerifierTransaction, the component group is represented as

$$A_i \subset (\{c_{i,1}, \ldots, c_{i,m}\} \cup \{h^0_{i,1}, \ldots, h^0_{i,m}\}),$$
$$\text{s.t. if } c_{i,j} \in A_i \text{ then } h^0_{i,j} \notin A_i \text{ and if } h^0_{i,j} \in A_i \text{ then } c_{i,j} \notin A_i.$$

Then, the ordered list K^0_i of transaction component for each subtree is calculated as

$$K^0_i = (k^0_{i,1},\ \ldots,\ k^0_{i,m})$$
$$\text{s.t. } \forall k^0_{i,j} = \begin{cases} h^0_{i,j} & \text{if } h^0_{i,j} \in A_i, \\ \textbf{Blake2s}(n_{i,j}||c_{i,j}) & \text{otherwise.} \end{cases}$$

Once K^0_i is computed for a component group, the rest of the operations, i.e., computations on the upper levels of the subtree and the main tree, is the same as the unfiltered transaction. The only difference is that, at the level $d = 1$, the Blake2s hash values computed on the values $k^0_{i,j}$ instead of $h^0_{i,j}$ as $h^1_{i,j} = \textbf{Blake2s}(k^0_{i,2j} \parallel k^0_{i,2j+1})$, such that the ordered list of $h^1_{i,j}$s is

$$H^1_i = (h^1_{i,1}, \ldots, h^1_{i,\lceil \frac{m}{2} \rceil}).$$

[2] `https://www.keylength.com/en/4/`

5 Transaction Backchain Validation

During the transaction validation, the prover should prove that the input and reference states in the transaction have a valid history and they have not been spent yet. In this section, we explain how this validation can be performed in a zero-knowledge manner in ZKFlow.

In the original Corda protocol, the input states to a transaction are identified by a `StateRef`. This is a combination of the transaction id of the transaction that created that state and its index in the output list of that transaction. A `StateRef` enables a transaction verifier to resolve the chain of previous transactions, so that the verifier can fetch and verify all of them. This verification allows the verifier to confirm that the contents of each `TransactionState` that is pointed to by the input or reference states of the current transaction are valid. The `TransactionState`s, i.e., the output states, of a transaction that are pointed to by the input or reference states of subsequent transactions are called the unspent transaction outputs (UTXOs) of that transaction and their contents can be used as the inputs of the subsequent transaction verification.

Revealing the content of `TransactionState`s in the transaction backchain to other parties is an important privacy concern in Corda. For a party that is a participant in the current transaction, it is legitimate to have access to the content of relevant `TransactionState`s of the current transaction. Similarly, these participants should also have access to the UTXO contents of the current transaction to be able to validate it. However, granting participants visibility of the full transaction backchain is not desirable for the participants that were involved in the historical transactions due to the leakage of their potentially sensitive data to future participants in the transaction chain. In ZKFlow, we overcome this privacy problem by including the validation of the backchain in the ZKP along with the validation of transaction id.

5.1 Zero-Knowledge Backchain Validation

As illustrated in Figure 4, the output states of a WireTransaction are `TransactionState` objects. Its input and reference states, on the other hand, are `StateRef` objects, each of which points to a `Transaction State` of a previous transaction. To prove the validity of a transaction's history, the prover must prove that the `Transaction State` from the respective `StateRef` is valid by computing the leaf hash of the `Transaction State` in the corresponding transaction.

In the validation of the transaction backchain in ZKFlow, the initiator, counterparty, and the notary have the same view of the transaction

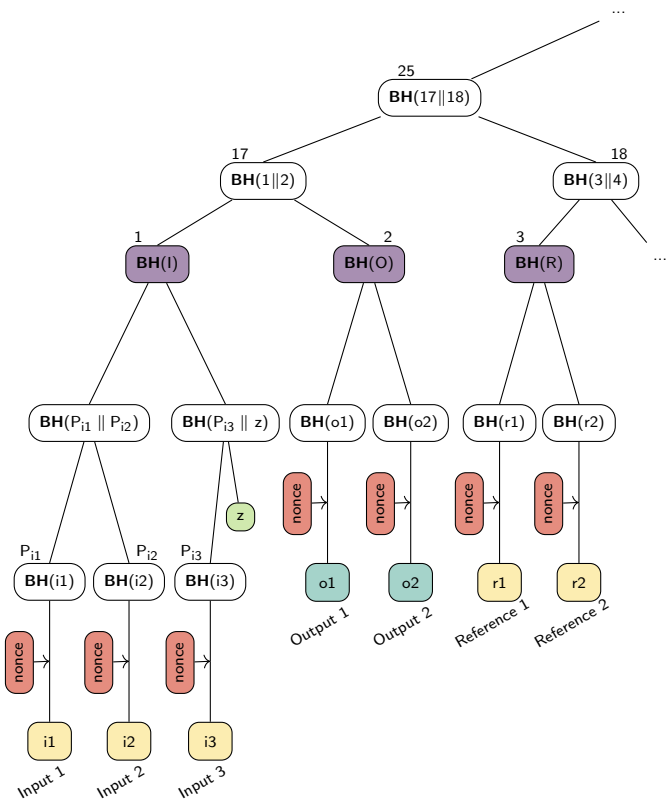

Figure 4: As a reminder, representation of a single transaction with its Transaction states and StateRefs in a standard WireTransaction, where everything is visible.

backchain. Therefore, they perform the validation in the same way. We illustrate the view of the initiator and the counterparty in Figure 5 and the view of the notary in Figure 6, respectively. Both figures illustrate the resolution of the backchain based on a single input state for simplicity. However, in the original protocol, the chain resolution is performed for each input and reference state of every transaction in the backchain.

As the participants of the current transaction, the initiator and the counterparty can view the content of the output states of the current transaction. Similarly, they also have access to the content of UTXOs, which are pointed to by the input and reference states of the head transaction and to the nonces of these UTXOs. The UTXOInfo that are used by the initiator and the counterparty are stored in arrays $\mathcal{I}$ and $\mathcal{R}$ for input and reference states, respectively. For each state, the UTXOInfo contains a `StateRef`, a `TransactionState`, and the nonce that is used in the computation of the leaf

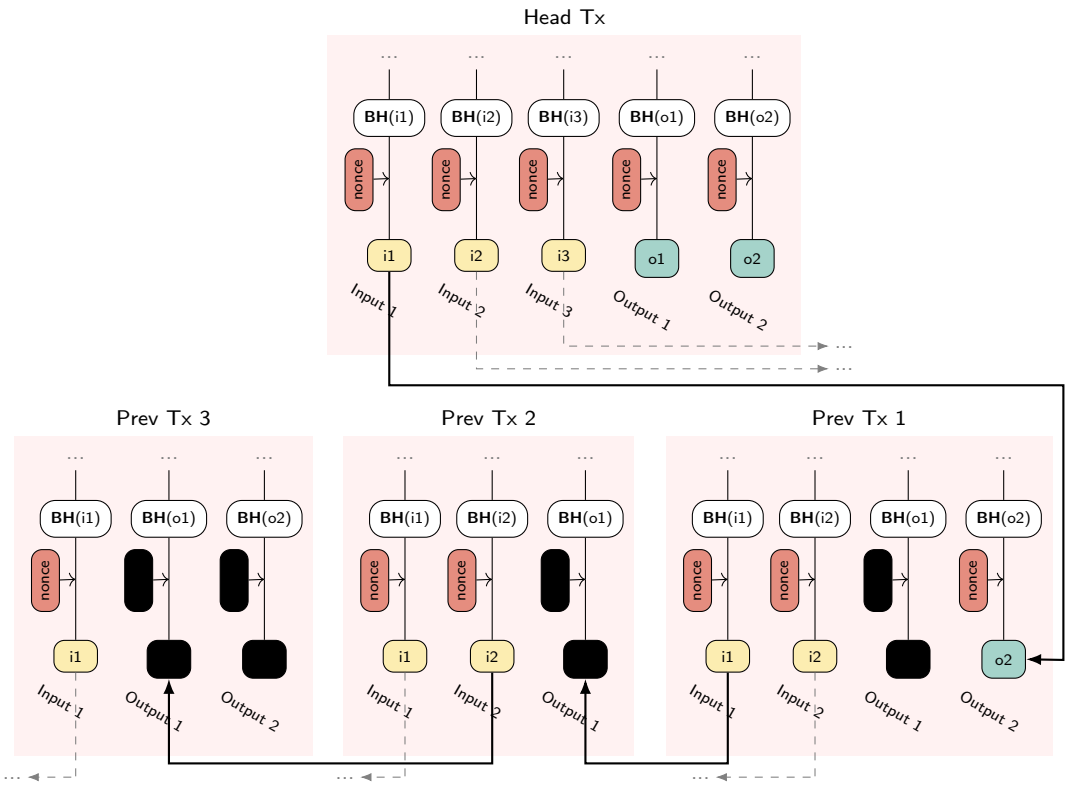

Figure 5: Initiator and counterparty's view of backchain.

hash of that state. The state information in the UTXOInfo is used for the validation of the head transaction, and it has no relation to the backchain.

The initiator has gathered all related UTXOInfo before transaction creation from their local storage or requested it from the counterparty. The counterparty, on the other hand, receives the UTXOs, $\mathcal{I}$ and $\mathcal{R}$, along with the proposed transaction. Both initiator and counterparty do not have access to the content of all other historical output states in the transaction backchain and any other component whose visibility is set to private, unless they were involved in it. Instead, they have access to their leaf hash digest, whose validity can be checked by validating the ZKP of the transaction that consumes it. In ZKFlow, we implement `SendUtxoInfosFlow` and `ZKReceiveUtxoInfoFlow` to share the UTXO information between the initiator and the counterparty, to resolve the backchain, and to recalculate the output hashes which are used for backchain validation.

After receiving the UTXO information from the initiator, the counterparty first validates the state contents in $\mathcal{I}, \mathcal{R}$ since the proposed transac-

tion's validity is dependent on the contents of these states. To do so, the counterparty

1. resolves and validates the backchain of each UtxoInfo's StateRef in $\mathcal{I}, \mathcal{R}$ by verifying the corresponding ZKP that validates the UTXO hash by recomputing them for each output state pointed to by inputs or references of the corresponding transaction using their corresponding nonces.

2. verifies the integrity of each UTXOInfo's `TransactionState` by:

   (a) fetching the output hash pointed to by the UTXOInfo's `StateRef` from the ZKVerifierTransaction,

   (b) recalculating the output hash using the nonce and state contents from the UtxoInfo provided by the Initiator. If they are identical, the state UtxoInfo's state contents have not been tampered with.

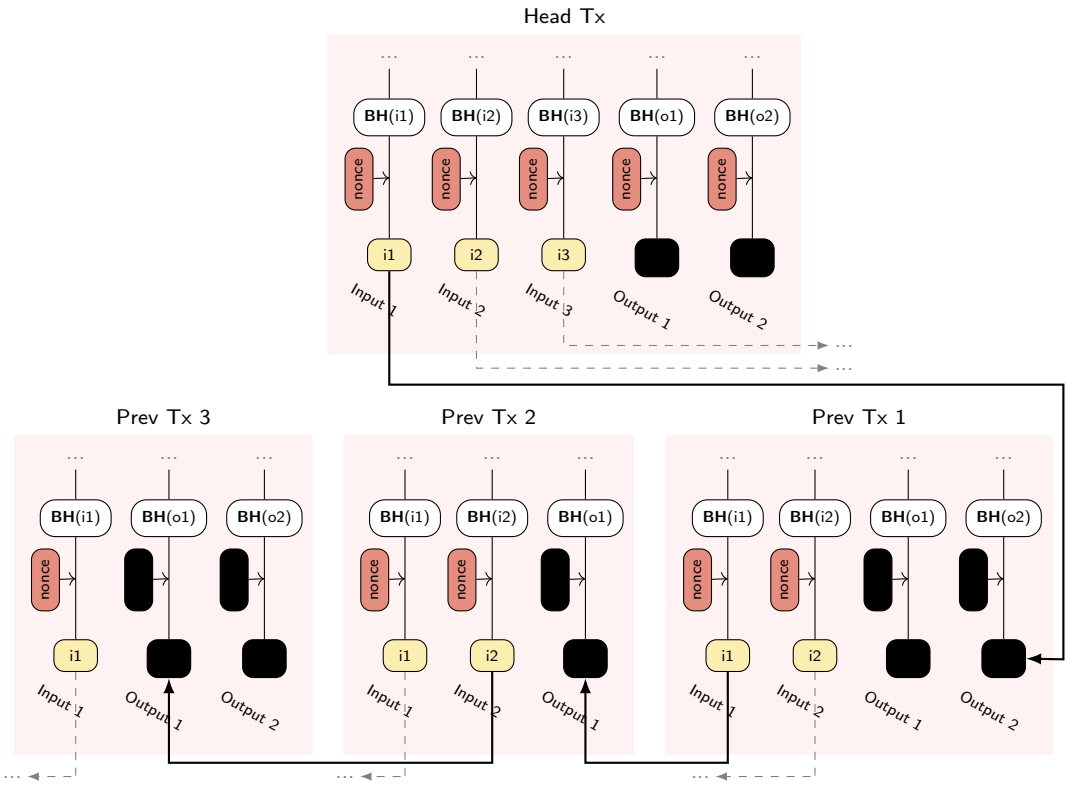

Figure 6: Notary's view of backchain.

The notary's view differs from the initiator and the counterparty since the notary has no access to the content of current transaction's output states nor the UTXOInfo. The notary can only see the corresponding component leaf hash for each output state in the transaction chain, including the current transaction. However, the validation of the backchain for the notary is the same as counterparties. Therefore, to validate the backchain of a new transaction, the notary should be able to verify the ZKP of each transaction in the backchain. A successful verification means that the public component leaf digest of an output state in the ZKVerifierTransaction is correctly computed from its private state content in the WireTransaction with its corresponding nonce value.

6 Zero-Knowledge Proof Statement

In this section, we explain the content of the zero-knowledge proof statement that is used in the validation of a transaction in ZKFlow. We follow the formatting style of ZKProof community reference document in our explanation [20].

Motivation: Perform a valid transaction on the Corda platform for a state transition such that the validation of the transaction does not reveal the content of the transaction to a third party.

Parties involved: There are three parties involved in the verification of the transaction: a notary, an initiator, and a counterparty.

- The initiator creates a WireTransaction, a matching ZKVerifierTransaction, and a zero-knowledge proof for each command that is a `ZKCommand` of the transaction to prove its validity of that ZKVerifierTransaction.
- The counterparty validates the WireTransaction, resolves and validates its private backchain (by verifying every historical ZKP in the backchain) and validates the UTXOInfo of the current transaction. If all are valid, the counterparty signs the transaction. The counterparty is not involved in the generation of the ZKP for the current transaction.
- The notary verifies the ZKVerifierTransaction that includes the zero-knowledge proof associated with the WireTransaction, resolves the transaction backchain and verifies every historical ZKP in the transaction backchain of the ZKVerifierTransaction.

What is being proved: The initiator proves that

- The WireTransaction is a valid transaction such that the component leaf hashes that are computed on its private and mixed-visible components are identical to the hash values provided in the corresponding public ZKVerifierTransaction and it satisfies the contract rules. By proving the equality of component leaf hashes, the initiator confirms that the transaction id computed as the Merkle root of the private WireTransaction is identical to the transaction id of ZKVerifierTransaction, which can be validated from public component hash values.

  Remark: The validation of a transaction in Corda also requires proving that the input and reference states of the transaction are not spent. In ZKFlow, this proof is performed as in the original Corda during notarization by a notary confirming that the visible input and reference states of the ZKVerifierTransaction not spent. Therefore, it is not included in the ZKP statement.

What information is needed by the verifier: The verifier needs access to

- The verifying keys VK for each ZKP associated with a `ZKCommand` to verify the WireTransaction is a valid transaction,
- The ZKVerifierTransaction to validate the public parts of the transaction.

Security goals: The transaction should be valid such that no malicious initiator can convince the verifier on the validity of a fake transaction.

Privacy goals: The validation of the transaction and its history by the counterparty and notary should not leak any unintended information to these parties apart from the fact that the transaction is valid. Therefore, their view of the transaction content is restricted.

Proof details: We use **bold** text style to represent **public** information, and *italic* text style to represent *private* information. On a high level, the initiator aims to prove the following statement:

> For the **Component Leaf Hashes**, **Input UTXO digests**, and **Reference UTXO digests**;

I know valid *Private and Mixed Components*, *Input UTXO states*, *Input UTXO nonces*, *Reference UTXO states*, and *Reference UTXO nonces*;

such that

- the component leaf hashes calculated from the *Private and Mixed Components* are identical to **Component Leaf Hashes**,
- the UTXO digests computed from the (*Input UTXO states*, *Input UTXO nonces*) and (*Reference UTXO states*, *Reference UTXO nonces*), which input and reference states of the *WireTransaction* point to, are identical to **Input UTXO digests** and **Reference UTXO digests**,
- *Private and Mixed Components* satisfies the contract rules for the corresponding `ZKCommand`.

In the proof of this statement, the initiator has the following information:

- Knowledge on how to create a transaction in a Merkle tree format and how to compute the component leaf hashes of the transaction from the leaves of the Merkle Tree,
- Knowledge on the chain of transactions that occurred before the current transaction.

To generate the proof, the initiator first

- gathers UTXOInfo for the inputs and references they want to use: *Input and Reference UTXO states* (`TransactionStates`) and *Input and Reference UTXO nonces*, **Input and Reference UTXO digests**,
- generates a *WireTransaction* between the initiator and the counterparty which is represented as a Merkle Tree and gathers its private and mixed components as *Private and Mixed Components*,
- calculates the **Component Leaf Hashes** of the *Private and Mixed Components*,
- computes a **ZKVerifierTransaction** which includes the public information about *WireTransaction* and the corresponding ZKPs for each `ZKCommand`. This transaction has the same id as the *WireTransaction*, since its Merkle tree is a filtered version of the original.

The prover sends the proof statement in Statement 6.1 to the verifier(s).

Proof Statement 6.1

$\pi = ZKPoK\{(Witness:$
{Private and Mixed Components, Input UTXO States,
Reference UTXO States, Input UTXO Nonces, Reference UTXO Nonces},
Instance :
{Component Leaf Hashes, Input UTXO Digests,
Reference UTXO Digests}) :
Predicate :
- `CalculateLeafHashes`(Private and Mixed Components) is identical to Component Leaf Hashes,
- `ComputeUTXODigests`(Input UTXO, Reference UTXO, Input Nonces, Reference Nonces) is identical to Input UTXO Digests and Reference UTXO Digests,
- `ValidateContractRules`(Private and Mixed Components) returns true.
}

7 ZKFlow

In the previous sections, we explained several building blocks that enables us to transform Corda's transaction validation mechanism to a privacy preserving one using ZKFlow protocol. Specifically, we described the transaction structure in Corda and how ZKFlow modifies Corda transactions to protect the confidential information. Next, we explained how a party can validate the backchain of a transaction and the related UTXOInfo for that transaction without revealing any sensitive information about it. Finally, we defined the content of the zero-knowledge proof that a prover can use to prove the validity of a transaction,

In this section, we explain how the participants interact with each other to validate a Corda transaction confidentially using these building blocks. As illustrated in Figure 1, the ZKFlow protocol consists of three main subprotocols, which are

- transaction signing and backchain validation,
- transaction notarization,

- transaction storage and broadcast subprotocols.

In the following, we explain how each subprotocol works.

7.1 Transaction Signing and Backchain Validation

The first phase of ZKFlow is the validation and signing of a newly built transaction by counterparty(s). In this phase, the counterparty validates both the new transaction and its transaction backchain. If the validation succeeds then the counterparty signs the transaction. Figure 7 shows the steps of signing and validation protocol performed by the initiator and the counterparty.

The subprotocol starts after the initiator creates a WireTransaction. To create the WireTransaction, the initiator should have already obtained all the states included in the transaction from their storage or request them from the counterparties. The initiator should validate the backchain of each state. Therefore, for each state, the initiator checks whether they have already validated their backchain and, if not, then the initiator resolves the backchain of the state and validates it. We explain how the initiator (and other participants) validates the backchain of transaction in Section 5 and the transaction structure in Section 4.

Once an initiator builds the WireTransaction, they sign its transaction id, with their private key and append the signature to the WireTransaction, which forms a SignedTransaction. Subsequently, the initiator performs standard Corda checks on the SignedTransaction for the transaction validity. Counterparties should have access to the contents of the input and reference states of the current transaction to validate it. Therefore, the initiator collects the UTXOInfo (which is a tuple of `TransactionState`, `StateRef`, and the nonce) for each state that is pointed by the input and reference states of the head transaction in two separate arrays as $\mathcal{I}, \mathcal{R}$. After that, the initiator sends the SignedTransaction and the UTXOInfo $\mathcal{I}, \mathcal{R}$ to the counterparty.

Upon receiving the proposed SignedTransaction and the UTXOInfo, the counterparty first validates the state contents in $\mathcal{I}, \mathcal{R}$, and resolves and validates the backchain of each state as explained in Section 5. As the next step, the counterparty validates the content and smart contract of SignedTransaction and verifies the initiator's signature. If all validations succeed, then the counterparty signs the transaction id of the SignedTransaction and sends it to the initiator.

Importantly, in this subprotocol, the ZKP of the proposed transaction is not shared with the counterparty since the counterparty already has access to the sensitive content. The proof generation is only handled by the initiator

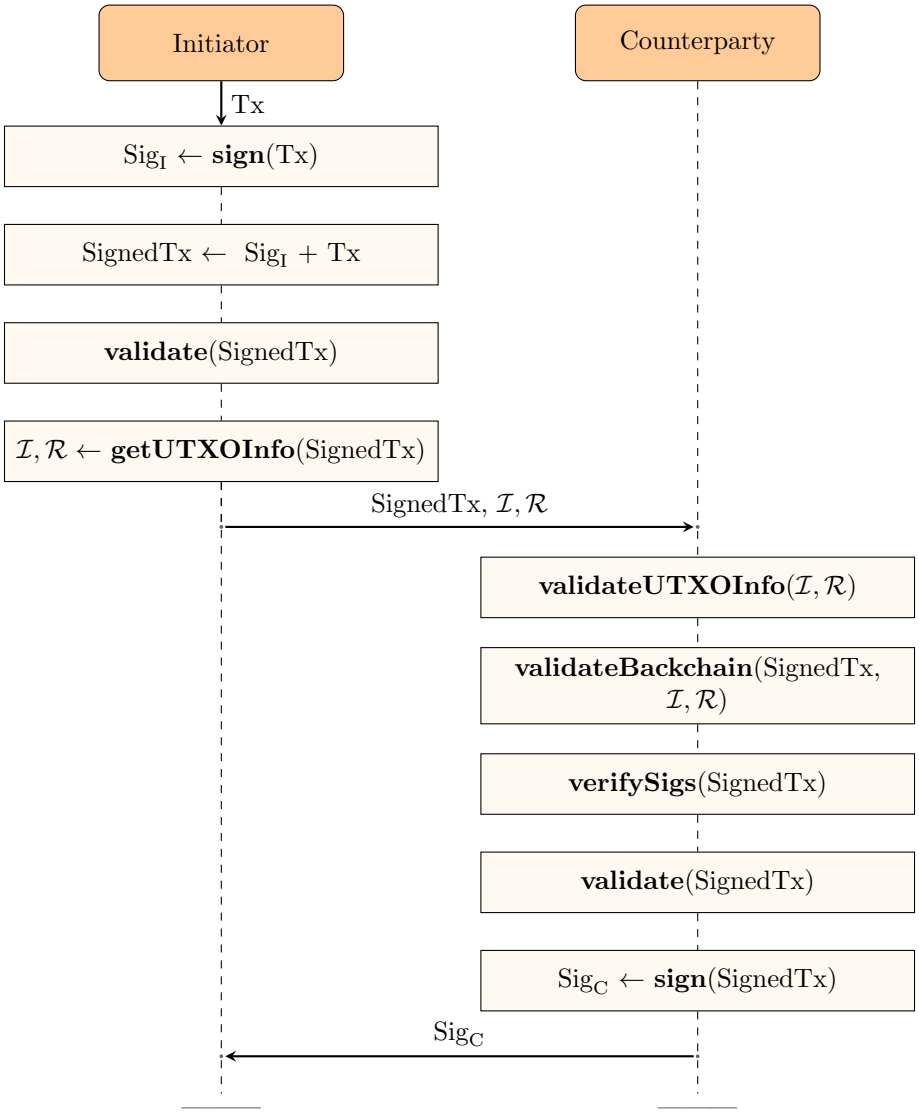

Figure 7: ZKFlow transaction signing and backchain validation subprotocol.

in the notarization of the transaction. The counterparty validates the ZKP that corresponds to the SignedTransaction later in the transaction storage phase. Sharing the ZKP with the counterparty in a later stage, does not degrade the security of ZKFlow, since the counterparty can relate the proof to the transaction and detect any inconsistencies between them.

7.2 Transaction Notarization

Once the initiator has collected the counterparty signature(s), the second phase is the notarization of the transaction. In the notarization subprotocol, a notary validates the transaction from its publicly available content and its ZKP. In Figure 8, we illustrate the steps of notarization in ZKFlow protocol.

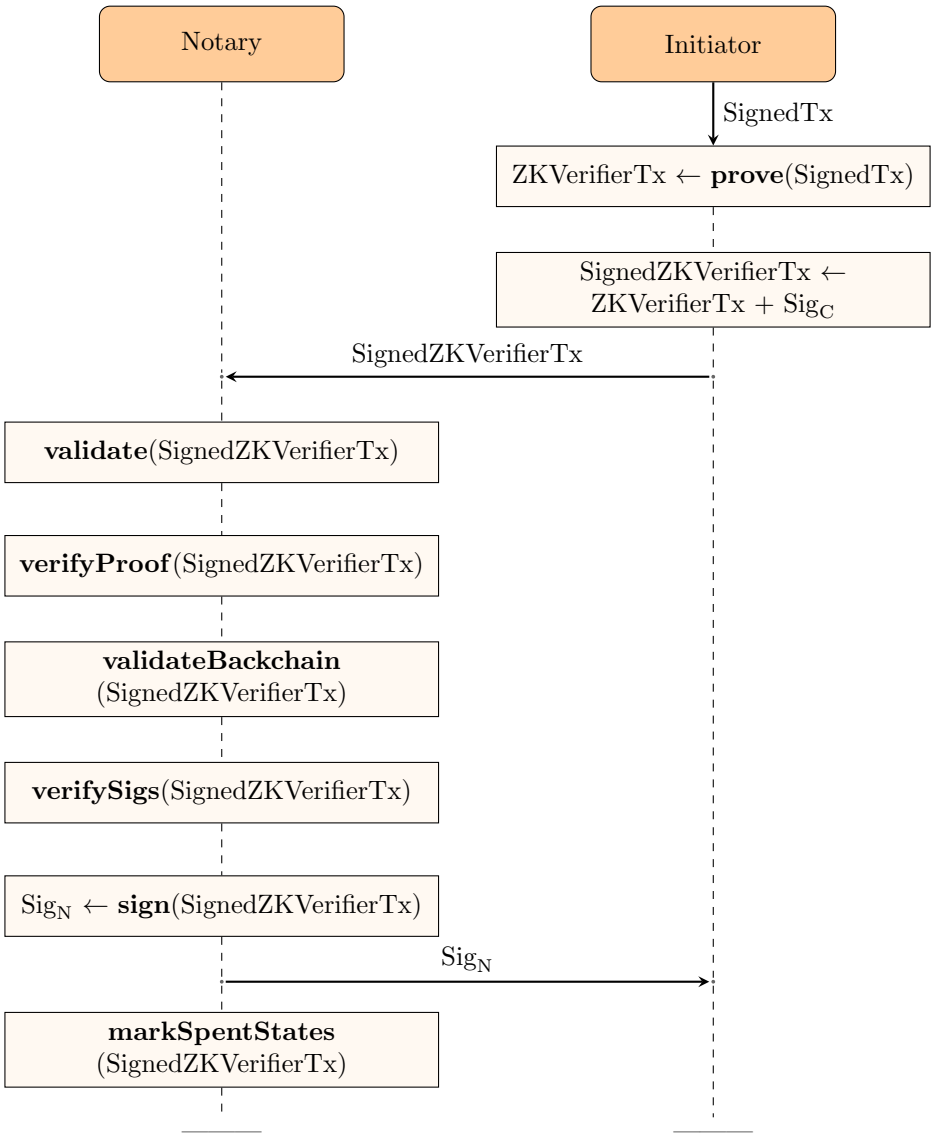

Figure 8: ZKFlow transaction notarization subprotocol.

The notarization subprotocol starts with the generation of ZKVerifier-Transaction by the initiator. A ZKVerifierTransaction is a public transaction

structure that contains a filtered transaction of the private SignedTransaction and a ZKP that is generated from the SignedTransaction. The initiator appends the signatures collected from the counterparties to the ZKVerifierTransaction and sends it to the notary as a SignedZKVerifierTransaction.

After receiving the notarization request from the initiator, the notary first validates the SignedZKVerifierTransaction by performing standard Corda checks. As a next step, the notary verifies the ZKP in the SignedZKVerifierTransaction which confirms the leaf hash of the components with private and mixed-visibility in the private SignedTransaction corresponds to the public leaf hash values in the SignedZKVerifierTransaction. Furthermore, the notary should perform the verification of the ZKPs in the transaction backchain with other necessary validity checks. Once all validations passed successfully, the notary verifies the signatures of parties involved in the transaction and signs the transaction id of SignedZKVerifierTransaction. The notary sends their signature to the initiator for the finalization of the transaction. Finally, the notary marks the input and reference state identifiers of the notarized transaction as **spent** in its state identifiers list to prevent future double-spent attempts.

7.3 Transaction Storage & Broadcast

The last phase of the ZKFlow protocol is the transaction storage and broadcast. If the proposed SignedTransaction is validated by the notary and the counterparties, it means that

- the ZKP of the transaction verifies that the leaf hash computation of the private SignedTransaction corresponds to the public leaf hashes of the SignedZKVerifierTransaction,
- the transaction has a valid backchain for each of its input and reference states that both the counterparty and the notary can confirm by validating the transaction backchain in a zero-knowledge way,
- the transaction satisfies the contract rules required for that transaction.

When the initiator receives the notarization approval from the notary, the initiator appends the notary signature Sig_N and verifies the signature. Furthermore, the initiator performs several verification operations on the SignedTransaction and SignedZKVerifierTransaction to validate that they truly belong to the same transaction structure. If the validation succeeds the initiator records the transaction into their local storage and sends the SignedTransaction and SignedZKVerifierTransaction to the counterparty. The counterparty

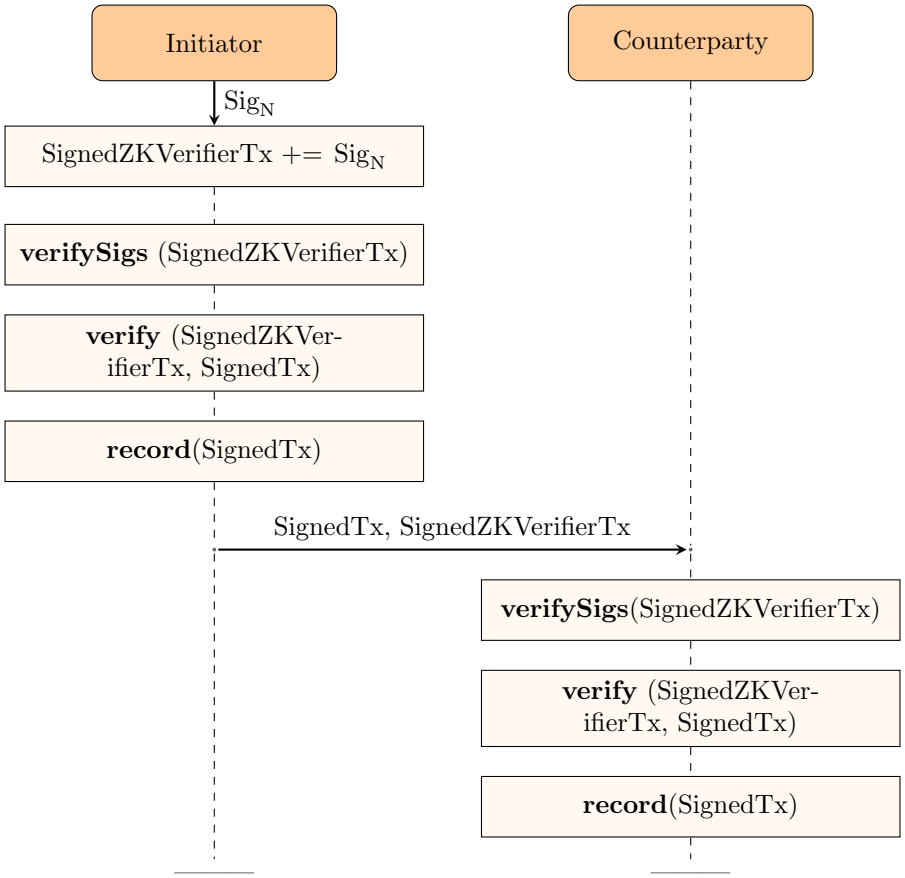

Figure 9: ZKFlow transaction storage and broadcast subprotocol.

performs the same checks on the SignedTransaction and SignedZKVerifier-Transaction and finally stores the transaction in their local storage.

8 ZKP setup and circuit management

In the previous sections, we explained how a transaction and its backchain can be validated using ZKPs and what is included in a proof. However, we have not explained how the proving and verifying keys are generated and managed. As discussed in Section 3.1, to perform proving and verification in ZKFlow, we need a pair of proving and verification keys PK and VK, respectively. Furthermore, we need information about the computation for which proof was generated. In ZKP terminology, it is often called"circuit" information since many protocols use arithmetic circuits to represent the computation. Altogether, this set of information is referred to as ZKP artifacts. In this

section, we explain procedures for generation (a.k.a. setup), distribution, and management of the ZKP artifacts.

8.1 Generation of artifacts

Many ZKP schemes require a trusted setup to generate artifacts. In a trusted setup, the information used or produced as intermediate output to generate the artifacts might enable an attacker to create fake proofs if revealed. Thus, the information, which is sometimes called "toxic waste"[3] should be destroyed immediately after setup. Any party that has access to this toxic waste can create fake proofs.

There are several ways of dealing with the trusted setup. In the following, we explain two methods of trusted setup, which are

- setup performed by a single trusted party,
- setup performed by a group of independent participants as a multiparty computation.

Depending on the trust proposition of the use case that ZKFlow is used for, one of the methods can be chosen for the setup.

8.1.1 Setup performed by a trusted party

The simplest and most centralized way of the trusted setup is to assign a trusted party to perform the procedure. The trusted party can generate the artifacts as specified and destroy toxic waste immediately afterward. Trust in this party may be derived from the financial or reputational losses that the party will suffer in case of misbehavior. The trust enforcement can be formalized with legal agreements, similar to how a denial-of-state [13] attack could be resolved. After setup, the trusted party signs the artifacts by their private key for future use.

Currently, ZKFlow uses this setup strategy since ZKFlow is expected/intended to run in a permissioned/closed network environment. Part of this closed environment is a Notary service, which is responsible for transaction validation, uniqueness, and ordering. By definition, Notary is a trusted service, which has the power of censorship to enable frontrunning or to allow duplicate transactions. Therefore, it is a suitable candidate to run this setup. An alternative for the trusted party is the party that builds and maintains the CorDapp, which uses ZKFlow.

[3] `https://z.cash/technology/paramgen/`

Depending on the use case that ZKFlow used for, the trust in centralized parties may not be acceptable, and strong security guarantees may be required. For such use cases, a setup performed in a decentralized setting by a group of participants is more suitable.

8.1.2 Setup performed as multiparty computation

A more complicated but secure way to generate ZKP artifacts is to perform an MPC (secure multiparty computation) ceremony, where all participants contribute to the process with some randomness. One of the most well-known examples of such MPC ceremony is the "Powers of Tau" ceremony used by ZCash[4]. In a multiparty setup, there is no central toxic waste. Instead, each participant has a personal fracture of toxic waste. Attackers can successfully generate fake proofs only if they can gather the fractures of toxic waste from all participants in the ceremony. It means that even if only one single party behaves honestly and is not compromised, generated artifacts are secure and safe to use.

8.2 Distribution of the circuit artifacts

Once setup is performed and artifacts are generated, we need to distribute them amongst participants in the network that use the CorDapp that operates with ZKFlow. In the current design of ZKFlow, there exists a set of artifacts per contract version. Thus, the distribution of artifacts is functionally very similar to Corda contract versioning, e.g., a party might need to fetch historical versions of artifacts when validating the backchain. However, a heavier contract upgrade procedure is required for producing states and proofs of the latest version. The heavier upgrade procedure is due to the distribution of PK, which might be as large as hundred megabytes to several gigabytes for complex contracts.

Because of the PK size, we cannot package its artifacts into the corresponding cordapp.jar. Instead, another distribution channel is required. In ZKFlow, we consider two different scenarios for the distribution of circuit artifacts:

- Installing new CorDapps,
- Verifying historical proofs during backchain validation.

In the following, we describe how each scenario works.

[4] `https://eprint.iacr.org/2017/1050`

8.2.1 Installing a new CorDapp

The requirement to have a PK for a specific version arises when a new version of CorDapp is delivered by the CorDapp developer. This procedure is not regulated by the Corda protocol. In ZKFlow, the PK should be distributed together with the corresponding CorDapp jar as a separate file. The only restriction is that the PK should be signed with the key used in Corda's `SignatureConstraint` for the CorDapp. Here, we assume it will be the key of the trusted party that performed the trusted setup. In the future, if MPC is used for the setup, this should rather be a group key that is distributed amongst the parties participating in a setup ceremony.

Scenario for CorDapp installation:

1. receive CorDapp jar (which includes VK and circuit) and PK from CorDapp developer or network operator
2. validate signatures for CorDapp and PK, they both should be signed by Corda network's `SignatureContraint` key
3. install CorDapp jar as usual in Corda
4. extract VK and circuit and put on a file system together with PK for future use.

Doing so gives users the ability to create new proofs and verify proofs for the current contract version. However, the users cannot verify proofs for historical versions of this contract if they did not install these specific versions of VK and circuit before. Such a situation might occur, for example, during the backchain validation, if historical transactions are created using historical versions of the CorDapp, meaning that also historical versions of ZKP artifacts were used.

8.2.2 Verifying historical proofs during backchain validation

In the backchain validation scenario, a party only requires the verification key and circuit. Conveniently enough, these artifacts are the smallest and can be easily packaged into a transaction as an attachment. Since Kotlin code is already packaged as `ContractAttachment`, ZKFlow just packages VK and circuits into the normal CorDapp jar.

In this case, the scenario for backchain validation would look as follows:

1. receive all required historical versions of the CorDapp jar from counterparty as contract attachment, same as how it is done in the base Corda protocol.

2. validate signatures on all of them, again, as in the normal Corda protocol.

3. extract VKs and circuits to be used in proof verification.

4. validate proofs using extracted artifacts.

For historical contract versions that were received during this process, it is possible to verify proofs. The same possibility does not hold to create new proofs since PK is not distributed due to its large size.

9 An Example Transaction Validation

In this chapter, we provide an example of transaction validation for ZKFlow. Our example considers a move transaction that contains two inputs, two outputs, two references, two attachments, two signers, and information about notary, time window, and network parameters.

9.1 Generating WireTransaction

As the first step, the transaction initiator, a.k.a., the prover, creates a WireTransaction with a Merkle tree structure and computes its Merkle root. Figure 10 shows the Merkle tree structure for the transaction. As mentioned earlier, the nonces and the hashes of the Merkle tree are generated using Blake2s hash function.

In the creation of transaction, the serialization used for each component group is important since the value of Merkle root, transaction id, is dependent on the content of the elements of component groups. Therefore, in the following, we explain how each node of the Merkle tree computed explicitly.

Computation of Nonce values The nonce values, which are represented with orange boxes on the lowest level of the Merkle tree in Figure 10 is computed as follows:

$$\text{nonce} = \textbf{Blake2s}(\text{privacySalt} \,||\, \text{groupIndex} \,||\, \text{internalIndex})$$

The groupIndex is defined in Corda as an enum named `ComponentGroupEnum`. The default index values in Corda, which are also used in this example is presented in Listing 1.

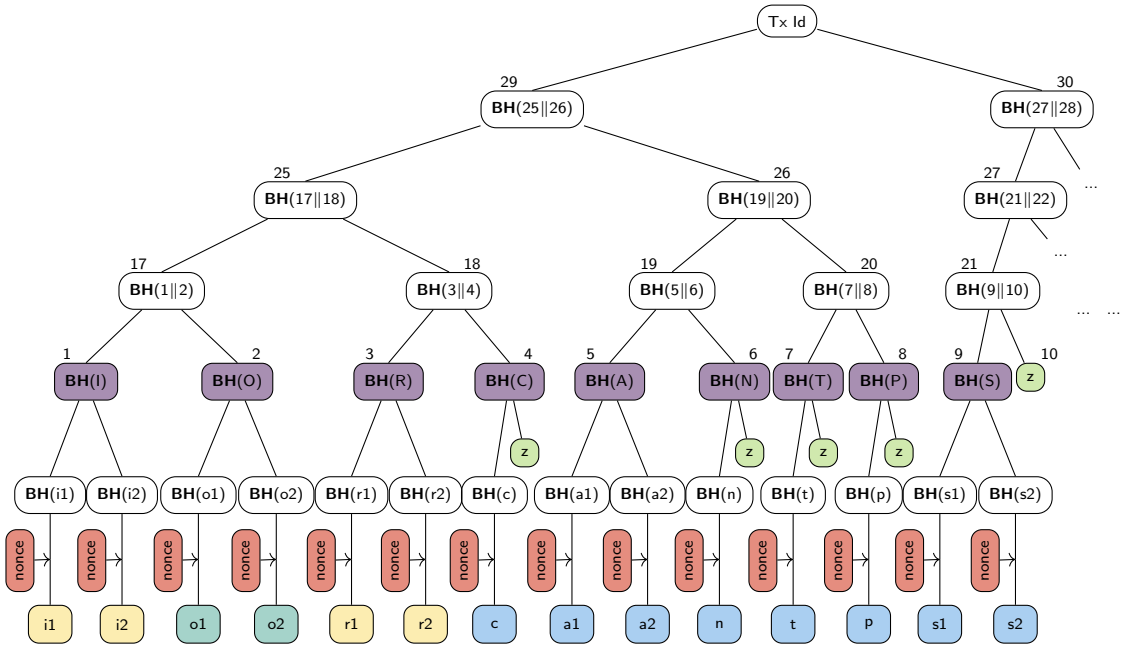

Figure 10: Merkle Tree representation of the example transaction.

```
enum ComponentGroupEnum {
    INPUTS_GROUP = 0,
    OUTPUTS_GROUP = 1,
    COMMANDS_GROUP = 2,
    ATTACHMENTS_GROUP = 3,
    NOTARY_GROUP = 4,
    TIMEWINDOW_GROUP = 5,
    SIGNERS_GROUP = 6,
    REFERENCES_GROUP = 7,
    PARAMETERS_GROUP = 8,
}
```

Listing 1: ComponentGroupEnum content.

The internalIndex is determined by the order of the elements within each component group which starts from 0. To assure collision resistance, the length of the preimage of the hash function should be fixed such that

- Privacy salt is 256 bits,
- groupIndex is 32 bits, and
- internalIndex is 32 bits.

If the number of bits in one of these fields is less than the required amount (most likely groupIndex and internalIndex), they should be prefixed with zeros as padded_value = $000 \cdots$ value.

Computation of Leaf Hashes In the computation of leaf hashes, depending on the structure of the component element the pre-image of the hash function differs. In the following, we explain how the leaf nodes are serialized for each component group.

- **Input Group** Each element of the input component group is a `StateRef`. In Listing 2, we provide the explicit content for these objects as they are used in this example. The type `NodeDigest` is the Pedersen hash digest of the transaction id of the transaction pointed by the `StateRef`, and `PubKey` is a public key object.

```
struct Party{
    owning_key: PubKey,
}

struct ZKContractState{
    owner: Party,
    value: i32,
}

struct TransactionState{
    data: ZKContractState,
    notary: Party,
}

struct StateAndRef{
    state: TransactionState,
    reference: StateRef,
}

struct StateRef{
    tx_hash: NodeDigest,
    index: i32,
}
```

Listing 2: The structure of the example `TransactionState`, `StateAndRef`, and `StateRef` objects.

The serialization of the input elements is performed on the `StateRef`. The leaf hash of an input element is computed on the serialized `StateRef` as

$$\mathbf{BH}(\mathrm{i}_j) = \mathbf{Blake2s}(\mathtt{nonce}_j \,\|\, \mathtt{tx\_hash}_j \,\|\, \mathtt{index}_j).$$

- **Output Group** Each element of the output component group is an object of type `TransactionState` that contains an example contract state `ZKContractState` as shown in Listing 2. The leaf hash of output elements is computed on the serialized `TransactionState`. In our example, we serialize a `TransactionState` as

$$\mathbf{BH}(\mathrm{o}_j) = \mathbf{Blake2s}(\mathtt{nonce}_j \,\|\, \mathtt{owner\_owning\_key}_j \,\|\, \mathtt{value} \,\|\, \mathtt{notary\_owning\_key}).$$

  Similar to the padding operation in the nonce generation, if the number of bits in the value field is less than 32, then it should be padded with zeros.

- **References Group** The structure of the references group and the computation of its leaf hashes is the same as input group. Thus, the leaf hash for each element in reference group us computed as

$$\mathbf{BH}(\mathrm{r}_j) = \mathbf{Blake2s}(\mathtt{nonce}_j \,\|\, \mathtt{tx\_hash}_j \,\|\, \mathtt{index}_j).$$

- **CommandData Group** In our example, we represent the command values in an enum structure as represented in Listing 3.

```
enum CommandData {
     CREATE = 0,
     MOVE = 1,
}
```

Listing 3: The content of the `CommandData` enum.

  In the example transaction, we use a single command in the transaction. The leaf hash for the command is computed as

$$\mathbf{BH}(\mathrm{c}) = \mathbf{Blake2s}(\mathtt{nonce} \,\|\, \mathtt{CommandData}),$$

  where the value of command data is a 32-bit integer (padded with zeros).

- **Attachment Group** An element of the attachment group is an object of type `AttachmentId` which is the result of the hash on the attachment documents (see Listing 4).

```
type AttachmentIdBytes = [u8; ATTACHMENT_ID_BYTES];

struct AttachmentId{
    value: AttachmentIdBytes,
}
```

Listing 4: The structure of the `AttachmentId` object.

The leaf hashes for attachment elements are computed on the serialized `AttachmentId` as

$$\mathbf{BH}(\mathrm{a}_j) = \mathbf{Blake2s}(\texttt{nonce}_j \,\|\, \texttt{AttachmentId\_value}_j).$$

- **Notary Group** In this example, the notary group contains a single element which is a `Party` object. The leaf hash for the notary is

$$\mathbf{BH}(\mathrm{n}) = \mathbf{Blake2s}(\texttt{nonce} \,\|\, \texttt{owning\_key}).$$

- **TimeWindow Group** The single element of the TimeWindow group contains a `TimeWindow` object as represented in Listing 5.

```
type TimeWindowBytes = [u8; TIME_WINDOW_BYTES];

struct AttachmentId{
    value: TimeWindowBytes,
}
```

Listing 5: The structure of the `TimeWindow` object.

The leaf hash for the serialized `TimeWindow` is computed as

$$\mathbf{BH}(\mathrm{t}) = \mathbf{Blake2s}(\texttt{nonce} \,\|\, \texttt{TimeWindow\_value}).$$

- **Parameters Group** The parameters group contains a hash digest that corresponds to the hash of network parameters. The leaf hash for `Parameters` is computed as

$$\mathbf{BH}(\mathrm{p}) = \mathbf{Blake2s}(\texttt{nonce} \,\|\, \texttt{Parameters\_value}).$$

- **CommandSigners Group** Each element of the signers group holds the public key of the corresponding `Party` that is involved in the transaction. The leaf hashes for signers group is computed as

$$\mathbf{BH}(\mathrm{s}_j) = \mathbf{Blake2s}(\mathrm{nonce}_j \,\|\, \texttt{owning\_key}_j).$$

Computation of Component Hashes Once the leaf hashes are computed, the next step is the computation of the group hashes for each component (represented as purple nodes in Figure 10). The group hashes for each

component is computed as follows:

$$\begin{aligned}
\mathbf{BH}(\mathrm{I}) &= \mathbf{Blake2s}(\mathbf{BH}(\mathrm{i}_1) \,\|\, \mathbf{BH}(\mathrm{i}_2)) \\
\mathbf{BH}(\mathrm{O}) &= \mathbf{Blake2s}(\mathbf{BH}(\mathrm{o}_1) \,\|\, \mathbf{BH}(\mathrm{o}_2)) \\
\mathbf{BH}(\mathrm{R}) &= \mathbf{Blake2s}(\mathbf{BH}(\mathrm{r}_1) \,\|\, \mathbf{BH}(\mathrm{r}_2)) \\
\mathbf{BH}(\mathrm{C}) &= \mathbf{Blake2s}(\mathbf{BH}(\mathrm{c}) \,\|\, 256\text{-bit zero}) \\
\mathbf{BH}(\mathrm{A}) &= \mathbf{Blake2s}(\mathbf{BH}(\mathrm{a}_1) \,\|\, \mathbf{BH}(\mathrm{a}_2)) \\
\mathbf{BH}(\mathrm{N}) &= \mathbf{Blake2s}(\mathbf{BH}(\mathrm{n}) \,\|\, 256\text{-bit zero}) \\
\mathbf{BH}(\mathrm{T}) &= \mathbf{Blake2s}(\mathbf{BH}(\mathrm{t}) \,\|\, 256\text{-bit zero}) \\
\mathbf{BH}(\mathrm{P}) &= \mathbf{Blake2s}(\mathbf{BH}(\mathrm{p}) \,\|\, 256\text{-bit zero}) \\
\mathbf{BH}(\mathrm{S}) &= \mathbf{Blake2s}(\mathbf{BH}(\mathrm{s}_1) \,\|\, \mathbf{BH}(\mathrm{s}_2))
\end{aligned}$$

As explained in Section 4, if there is one element in the component group then that element is assigned as the component group hash following Corda's current Merkle tree algorithm.

Computation of the Root Hash The last step is the computation of the Merkle root from the component group hashes as follows

$$\begin{aligned}
\mathbf{BH}(17) = \mathbf{BH}(1+2) &= \mathbf{Blake2s}(\mathbf{BH}(\mathrm{I}) \,\|\, \mathbf{BH}(\mathrm{O})) \\
\mathbf{BH}(18) = \mathbf{BH}(3+4) &= \mathbf{Blake2s}(\mathbf{BH}(\mathrm{R}) \,\|\, \mathbf{BH}(\mathrm{C})) \\
\cdots \quad &\cdots \\
\mathbf{BH}(30) &= \mathbf{Blake2s}(\mathbf{BH}(27) \,\|\, \mathbf{BH}(28)) \\
\text{Tx Id} &= \mathbf{Blake2s}(\mathbf{BH}(29) \,\|\, \mathbf{BH}(30))
\end{aligned}$$

These steps can also be verified from Figure 10. After creating the WireTransaction, the initiator performs the transaction signing and backchain validation subprotocol with the counterparty.

9.2 Generating ZKVerifierTransaction

Once the WireTransaction is validated by the counterparty, the next step is its notarization. As explained in Section 7, to hide the privacy-sensitive content of the WireTransaction, ZKVerifierTransaction is used in the notarization. The ZKVerifierTransaction contains the filtered WireTransaction and its zero-knowledge proof. In the filtered transaction, the contents of several component groups are hidden. Different from the original Corda, for the validation of backchain the leaf hash of output elements are provided in the ZKVerifierTransaction. Also, the signers keys are included in the ZKVerifierTransaction since the signature validation is moved out of the ZKP circuit. Figure 11 illustrates the filtered transaction for the example transaction.

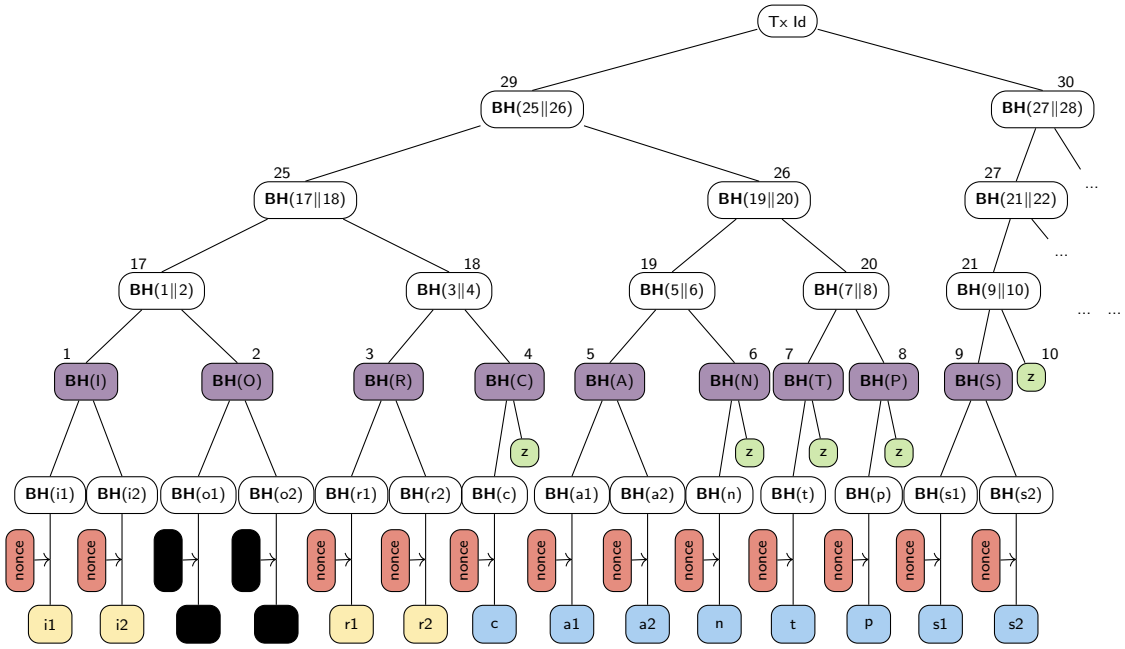

Figure 11: Merkle Tree representation of the example ZKVerifierTransaction

9.3 Generating ZKP

After preparation of the WireTransaction and the ZKVerifierTransaction, as the next step, the initiator generates the zero-knowledge proof. As explained previously, the visibility of output states and input and reference utxos are set private by default and the rest of the components are publicly visible. Therefore, the witness of the proof contains the private output states of WireTransaction and the input and reference utxos that are the nonces of the leaf hashes of the unspent transaction outputs. The JSON file that shows the witness structure for the example transaction is as follows:

```
"witness": {
    "outputs": [ ... ],
    "input_utxo_nonces": [ ... ],
    "reference_utxo_nonces": [ ... ],
    "input_utxos": [ ... ],
    "reference_utxos": [ ... ],
    "privacy_salt": [ ... ],
}
```

Listing 6: Witness of the ZKP for the example transaction.

The corresponding public input of the proof that contains the leaf hashes for

the output states and the computed UTXO digests are structured as follows:

```
"public_input": {
    "output_hashes": [ ... ],
    "input_utxo_hashes": [ ... ],
    "reference_utxo_hashes": [ ... ],
}
```

Listing 7: Instance of the ZKP for the example transaction.

Once the ZKVerifierTransaction is sent to the notary, the validation steps are performed by the notary as explained in Section 7.

10 Discussion

Zero-knowledge proofs provide strong security guarantees to protect sensitive data. Despite their strong security guarantees, adopting ZKPs to an existing platform is not trivial. Rather it requires careful consideration regarding performance, flexibility, and compliance.

10.1 Performance

One concern in the adoption of ZKPs is their performance, which is related to the circuit size. The circuit size is affected by both the number of component elements and the size of each element. Thus, ZKPs cannot provide efficient setup and proving time for transactions with hundreds of input or output states. Similarly, we recommend not to include large documents as attachments within the circuit.

10.2 Standardization

Another concern in the usage of ZKPs is the lack of standardization. So far, there is no international standard regarding the usage of ZKPs or ZKP-friendly cryptographic primitives. However, the extensive research and development effort around the adoption of ZKPs eliminates this concern by providing an independent review of the existing schemes from multiple resources with respect to their security and performance. Furthermore, the standardization effort for ZKPs by the ZKProof community, which is also supported by NIST [17], enhances the reliability of the cryptographic scheme.

10.3 Trusted setup

A final concern related to ZKPs is their flexibility when the ZKP scheme does not support a universal or updateable setup. Any change in the smart contract requires setting up the ZKP circuit anew, but it also requires redistributing the new proving and verifying keys to the respective parties as explained in Section 8. Besides that, technically, this step is the most time-consuming part of ZKPs. However, the research on ZKPs is rapidly developing, and several trends might influence future artifacts setup and distribution procedures. In the following, we discuss some of the trends.

10.3.1 Schemes without trusted setup

There exist ZKP schemes that do not require a trusted setup. It means that there is no toxic waste, no party that can forge proofs, and the setup procedure is greatly simplified. Bulletproofs [6] and zkSTARKs [5] are examples of such schemes. Although these schemes are behind zkSNARKs on the performance side and require some battle-testing, they are actively developing and can be proved useful in the upcoming future.

10.3.2 Schemes with universal setup

For many schemes, setup is required per every code version, i.e., each version of each Corda contract. Yet some schemes allow to make a setup once and then reuse its output in generating proofs for any contracts. This feature greatly simplifies both setup and distribution as only one set of artifacts exist on the network, which can be distributed only once, for example, with the Corda itself.

10.3.3 TinyRAM

TinyRAM [3] is a ZKP virtual machine that, in addition to private data, also takes as input parameters the assembly code of smart contract to be executed, i.e., proved/verified. In this case, only one version of a ZKP circuit exists, and the setup should be performed only once. Additionally, for platforms targeting Ethereum or any other public ledger, a distributed and secure setup ceremony is performed for its master contract on the ledger's main network and it might be possible to reuse their artifacts in the Corda setting. This feature can reduce any trust requirement for ZKFlow or Corda network operators to the very minimum, as any party will always be able to compare their artifacts or use Ethereum ones.

10.3.4 Smart contract code compilation

Currently, in ZKFlow smart contracts should exist in two forms: as Kotlin code and as chosen ZKP circuit generation framework code. ZKFlow enhances developer experience by pre-generating a large infrastructural part of a ZKP code, but there are possibilities for further improvement. If ZKFlow can produce circuits directly from Kotlin smart contract code, then the distribution of artifacts can be discarded in combination with techniques from Section 10.3.2 and 10.3.3. Kotlin compiler backend has a modular multiplatform structure that allows for such compilation.

11 Conclusion

In this paper, we presented the ZKFlow protocol that addresses the privacy problems in the Corda ledger during transaction validation. Our protocol uses zero-knowledge proofs to achieve data protection in a verifiable setting due to their provable security guarantees. We explained how new transactions are introduced to enable the usage of ZKPs in Corda. We showed that our protocol aims to adhere to the original Corda design and adds new structures only if necessary.

Acknowledgements

We would like to thank Mike Hearn and Kostas Chalkias for their valuable comments and feedback in the preparation of this document.

References

[1] Jean-Philippe Aumasson. Too much crypto. *IACR Cryptol. ePrint Arch.*, page 1492, 2019.

[2] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5. *IACR Cryptol. ePrint Arch.*, 2013:322, 2013.

[3] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for C: verifying program executions succinctly and in zero knowledge. In *CRYPTO (2)*, volume 8043 of *Lecture Notes in Computer Science*, pages 90–108. Springer, 2013.

[4] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *USENIX Security Symposium*, pages 781–796. USENIX Association, 2014.

[5] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptol. ePrint Arch.*, page 46, 2018.

[6] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Gregory Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *IEEE Symposium on Security and Privacy*, pages 315–334. IEEE Computer Society, 2018.

[7] Corda. Corda v hyperledger v quorum v ethereum v bitcoin. `https://www.corda.net/blog/corda-v-hyperledger-v-quorum-v-ethereum-v-bitcoin/`, 2019.

[8] Corda. Confidential identities. `https://docs.corda.net/docs/corda-enterprise/4.6/cordapps/api-confidential-identity.html`, n.a.

[9] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *STOC*, pages 291–304. ACM, 1985.

[10] Peter Grad. Reports: Intel chips have new security flaws. `https://techxplore.com/news/2020-06-intel-chips-flaws.html`, 2020.

[11] Mike Hearn and Richard G. Brown. Corda: A distributed ledger. *Corda Technical White Paper*, 2019, 2016.

[12] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification version 2020.1.11 [overwinter+sapling+blossom+heartwood]. `https://github.com/zcash/zips/blob/master/protocol/protocol.pdf`, 2020.

[13] Tommy Koens, Scott King, Matthijs van den Bos, Cees van Wijk, and Aleksei Koren. Solutions for the corda security and privacy trade-off: Having your cake and eating it. *White Paper*, 2019.

[14] Yehuda Lindell. The security of intel sgx for key protection and data privacy applications. *Unbound Tech*, 2018.

[15] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ISCA*, page 10. ACM, 2013.

[16] Corda Network. Understanding corda network services. `https://corda.network/faq/frequently-asked-questions/#1-network-choice`, n.a.

[17] Rene Peralta, Luis T.A.N. Brandao, and Angela Robinson. Privacy-enhancing cryptography. `https://csrc.nist.gov/Projects/pec/zkproof`, n.a.

[18] R3. Enterprise blockchain success stories for every business in every industry. `https://www.r3.com/customers/insurance/`, n.a.

[19] Radu Sion. *Trusted Hardware*, pages 3191–3192. Springer US, Boston, MA, 2009.

[20] ZKProof. Zkproof community reference. `https://docs.zkproof.org/pages/reference/reference.pdf`, 2019.