INSTITUTO TECNOLÓGICO DE BUENOS AIRES – ITBA

ESCUELA DE INGENIERÍA Y GESTIÓN

ETHEREUM'S SCALABILITY SOLUTIONS FOR GAMING

Authors: Balaguer, Pedro (Leg. 55795)

Caracciolo Lopez, Juan Franco (Leg. 56382)

Garrigó, Mariano (Leg. 54393)

Tutor: Cortesi, Mariano

A THESIS SUBMITTED FOR THE DEGREE OF

SOFTWARE ENGINEERING

Place: Buenos Aires

Date: December 5, 2019

Contents

List of Figures

Abstract

Decentralized gaming is one of the newest blockchain related developments being researched nowadays since scalability solutions are required due to the inherit confirmation time and cost of blockchains. This project explores two of these and implements them in a proof-of-concept game. The first scalability solution, **Plasma**, allows players to freely send in-game assets to one another. An additional feature is proposed in which players are also able to trade these assets atomically. The second scalability solution, **Force Move Channels**, allows users to do off-chain computation in a **PVP** environment while relying in the blockchain for discrepancies. While these two solutions have their intrinsic limitations, a working prototype is proposed where users can purchase in-game assets and battle with them using minimal costs compared to previous **Ethereum**'s implementations.

Introduction

Before the concept of **Ethereum** existed, its author, *Vitalik Buterin*, was an avid **World of Warcraft** player, a popular MMORPG at the time. It was not until **Blizzard**, the company behind World of Warcraft, removed a component Vitalik owned in the game that he realized the issues that centralized services can bring [1]. Moving almost a decade to present time, the technology is getting closer to the idea of **decentralized gaming**, a platform in which players are complete owners of their in-game assets and can rest assure that the code being executed is fair for everyone.

1.1 Blockchain

Historically, when two parties want to make a transaction over the internet, they appeal to a *mediator* to act as an intermediary for the operation. The mediator's responsibilities involve tasks like keeping record of ownership and validate authenticity of items and participants. However, in order to choose a middleman, first both parties need to trust it, since it will have complete control over the transaction. This raises the following question: is there a way of conducting a transaction without an intermediary and without the need of trust? To answer that question, the concept of **blockchain** gets introduced.

The first mention of the notion of blockchain comes from the publication of Bitcoin's whitepaper in 2008 [1], where a person, or a group of persons, behind the pseudonym *Satoshi Nakamoto* detailed a *peer-to-peer electronic cash system* for online payments without intermediaries. The real innovative technology behind this payment system is what is today known as **blockchain**.

[1] `https://about.me/vitalik_buterin`

Blockchain is a technology where records of transactions are distributed among a network of computers, called **nodes**. Transactions are validated and added into **blocks**, which are then chained together, hence the name, giving these transactions a clear chronological order. After a block is created, it is distributed among the network for the rest of the nodes to validate. When a block is validated by the majority of the nodes, all the transactions inside it are officially **valid**, **immutable** and **sorted in time**.

In order to submit blocks, and avoid spamming, members in the network, called **miners**, must compete to validate transactions by solving complex coded problems, where the first to solve it receives a reward. Each time a block is validated, this problem changes, encouraging the network to move onto the next block and avoiding hoarding of transactions. Additionally, each transaction provides a fee which is distributed among miners as a reward for maintaining the network valid and secure.

To summarize, blockchains have the following properties:

Decentralization

Rather than having a unique trusted authority that centralizes all the records and is in charge of all validations, blockchains distribute the information across the network. In order to forge the blockchain, over 51% of the network processing power is needed, meaning the cost of achieving such computing power outweighs greatly the benefits gained from it. This prevents the network from being controlled by a single entity.

Immutability

Every record in a blockchain is permanent, any attempt to change the history will be rejected and any node deviating from the consensus will be excluded from the network.

Transparency

All transactions are recorded and stored in every node and any participant can see and check any transaction at any time.

1.2 Ethereum

Bitcoin pioneered the cryptocurrency world, taking the first steps and gaining traction because of it. But multiple developers saw further potential in **blockchain** and more cryptocurrencies emerged, many of them adding more capabilities to what Bitcoin offered. One of those blockchains is **Ethereum** [2].

Ethereum provides a **Turing-complete programming language** that enables the blockchain to retain a state and execute functions that may change it. This state is replicated all over the network, and any code executed is executed the same on every node, transitioning the **state machine** to the next state. After this, everyone should end up in the same state, and if a node fails to do so, it is excluded from the network.

All this capability is allowed by the creation of **Smart Contracts** that enclose executable code. Any regular account (pair of private/public key) can create or interact with these smart contracts. Just as any account, a smart contract also has a public address, however, it can't generate new transactions, only can react to them using its inmutable code.

When a program is executed, it must run on every node on the **Ethereum Virtual Machine** (EVM). The **EVM** has limited stack, volatile memory and permanent storage as any regular virtual machine does. The key difference is that the execution of bytecode inside the **EVM** is limited by the amount of **gas** provided. Gas is a fee that the caller of the execution must provide to pay the **miners** running the code, and will be consumed by the **EVM** for each instruction executed. If gas is completely consumed before reaching the end, then the operation is reverted, but if there is unused gas at the end of the execution it is returned to the caller account. Since every node must execute the code, providing gas is essential to prevent overloads or attacks to the network. Storage also consume gas because of the same reasoning, since to store anything in the blockchain every node in the network must store it.

Objective

2.1 Game concept

The purpose of this project is to showcase a working prototype of a decentralized game. **CryptoMonBattles** is a collectible game where players buy monsters, called **CryptoMons**, and then are able to **trade** and **battle** with them.

In **CryptoMonBattles** there are 151 different **CryptoMon** species to obtain and collect, however, each **CryptoMon** instance has an unique set of stats, like attack power, defense and speed, making each new **CryptoMon** different from the previous one, generating scarcity. **CryptoMons** with better stats will be more valuable since they are able to have the upper hand in a battle. Adding this to the ability to **trade** and **sell CryptoMons**, an economy can be created around these monsters.

When two players decide to **battle**, each one can decide which **CryptoMon** to use. **Cryptomon**'s species define a mechanism close to *Rock-Paper-Scissor* in which, for example, a *Fire* **CryptoMon** will have an advantage over a *Grass* one, but a *Grass* one will have an advantage over a *Water* one. This means that no sole **CryptoMon** can be better than the rest, as the situation in which they battle affects its odds of winning.

Once the players agree on two **CryptoMons** to battle with, a **bet** is made and the winner of the battle will be able to reclaim it. This gives **CryptoMons** a real-life value as better **CryptoMons** will make more earnings. The battle proceeds between the two players, taking turns making decisions, with some level of randomness involved, until one of the two **CryptoMons** can't continue battling, declaring the surviving **CryptoMon** the winner.

2.2 Ethereum Implementation

Building this application directly on top of **Ethereum**'s blockchain is possible since it provides a *Turing-complete programming language*, but it would be very inconvenient and costly. Firstly, for every transaction in the blockchain there is a fee that must be paid to miners, that means that the more a user makes transactions, the more money it will pay. These transactions involve purchasing a **Cryptomon**, transferring a **Cryptomon**, trading a **Cryptomon** with another user, initiating a battle, making a move in the battle, etc.

Secondly, **Ethereum** has scaling issues since the blockchain is only capable of processing 15 transactions per second. This may lead to long confirmation times and therefore slower response times. These confirmation times can range from 15 seconds to a couple a minutes. When playing a game, waiting for this can get tiring for players as, in battles consisting of multiple rounds, each move's confirmation time will add up to a considerable amount.

For these reasons, the following scalability solutions will be looked into:

Plasma

An off-chain solution for free **CryptoMon** transfer while keeping the accountability of the **Ethereum** blockchain.

Atomic Swaps

A feature built on top of **Plasma** to allow it to support the trade of two **CryptoMons** between two different users.

Force Move Channels

An off-chain **PVP** computation system where users can battle against one another while using **Ethereum** as a judge of any discrepancies.

Plasma

3.1 Introduction

Plasma is one of the early scalability solutions for **Ethereum**. It was conceptualized by *Joseph Poon* and *Vitalik Buterin* in 2017 [3]. Even though **Plasma** has gone through many iterations since its early concept, the initial idea still holds true. **Plasma** as a solution consists of a **SideChain** implementation operated by a private user (known as **the operator**). This operator can be viewed as an untrusty party, but that should not affect the functionality of **Plasma**. This **SideChain** consists of *UTXO-like* transactions that are added to a Merkle Tree. As the name describes *Unspent Transaction Outputs* means any transaction indicates spending a previous transaction, which can only be spent once. This new transaction becomes the latest *UTXO* waiting to be spent. A chain of these *UTXO* can be retrieved to know the history of some funds, and thus, the true and latest owner, as every transaction can be validated by the signature of the previous owner.

This is true on most blockchains, however, what sets apart a private-owned blockchain from a decentralized one is the non-forgeability of its history. In a decentralized blockchain, any particular individual looking to forge the past will be excluded from the main consensus and not be taken into account, while in a private operated one, the forgery could be accomplished. For this problem, **Plasma** proposes using a decentralized blockchain as a way to provide enforceability for a private blockchain: By providing the *RootHash* value of the transactions to the decentralized blockchain when a block is submitted, the transactions can be considered as consolidated, and any forgery in a past transaction of the private chain wouldn't match with the submitted value at the decentralized one. Also, this

considerably reduces the amount of data submitted to the decentralized blockchain, while still having its benefits.

3.2 Plasma Cash vs Plasma Debit

Plasma has suffered many changes during its lifespan. Many iterations have occurred, such as **Plasma MVP**[4], **Plasma Cash**[5] and **Plasma Debit**[6], each one with increased complexity. While **Plasma MVP** is an early concept using *UTXOs*, **Plasma Cash** and **Plasma Debit** are more robust, and the main difference is that **Plasma Cash** is limited to transactions of *Non-fungible tokens*[7], while **Plasma Debit**, still at early concept stages, allows the fungeability and merging of funds.

As this project is focused on the scalability of **Ethereum** for gaming purposes, fungeability of tokens is not required. This projects is limited to *Non-fungible tokens* as they can represent a wide arrange of assets inside many gaming scenarios. For this reason, **Plasma Cash** will be the main focus of the explanation and any future considerations will be limited to *Non-Fungible tokens*.

3.3 Plasma SideChain

As explained before, **Plasma** attempts to use a decentralized blockchain as a mean of enforceability. The first step would be to have a contract deployed in **Ethereum**, known as the **Founding Contract**. This contract will have many different tasks to enforce and validate everything happening off-chain. The first function the contract needs to support is the ability to receive and store *RootHashes* of blocks with their respective timestamp. This *RootHash* will be the result of a *Sparse Merkle Tree* (chapter 8) where the leaves are the transactions submitted to **the operator**. The collection of these rootHashses will be known as the **RootChain**. This results in 2 chains, the **SideChain**, managed by **the operator** in a private manner, and the **RootChain** which is in charge of storing the *RootHashes* for future verification, running on **Ethereum**.

```
Interface FoundingContract {
   RootChain::Map<BlockNumber, RootHash>
   function submitBlock(BlockNumber, RootHash) onlyOwner
   function validateInclusion(BlockNumber, slot, Proof)
   function deposit(ERC721Token) (section 3.4.1)
   function exit(slot, ExitData) (section 3.5)
   function challenge(slot, ChallengeData) (section 3.6)
}

Interface Operator {
   SideChain::Block[]
   function transfer(txBytes, signature) (section 3.4.2)
   function getHistory(slot)::Map<BlockNumber, Proof>
   function getExitData(slot)::ExitData (section 3.5)
}

Interface Block {            Interface Proof {
   BlockNumber                  txBytes  //null if no tx in block
   RootHash                     signature //null if no tx in block
   txBytes[]                    MerkleProof
}                            }
```

The **SideChain** is completely controlled by **the operator**. Decentralization of the **SideChain** could be achieved, but would not accomplish much since **Plasma** guarantees accountability even in a single node privately operated **SideChain**. With the *RootHash* saved in Ethereum, any user can make a **Merkle proof** against it to validate the inclusion of a transaction to a block. Once a *RootHash* is submitted there is no going back, as **Ethereum** can't be forged. This method comes with a clear advantage: cost reduction. Submitting a single hash is multiple times cheaper than submitting many transactions to **Ethereum**. What **Plasma** attempts to do is mitigate costs by grouping and condensing a number of transactions into a single value.

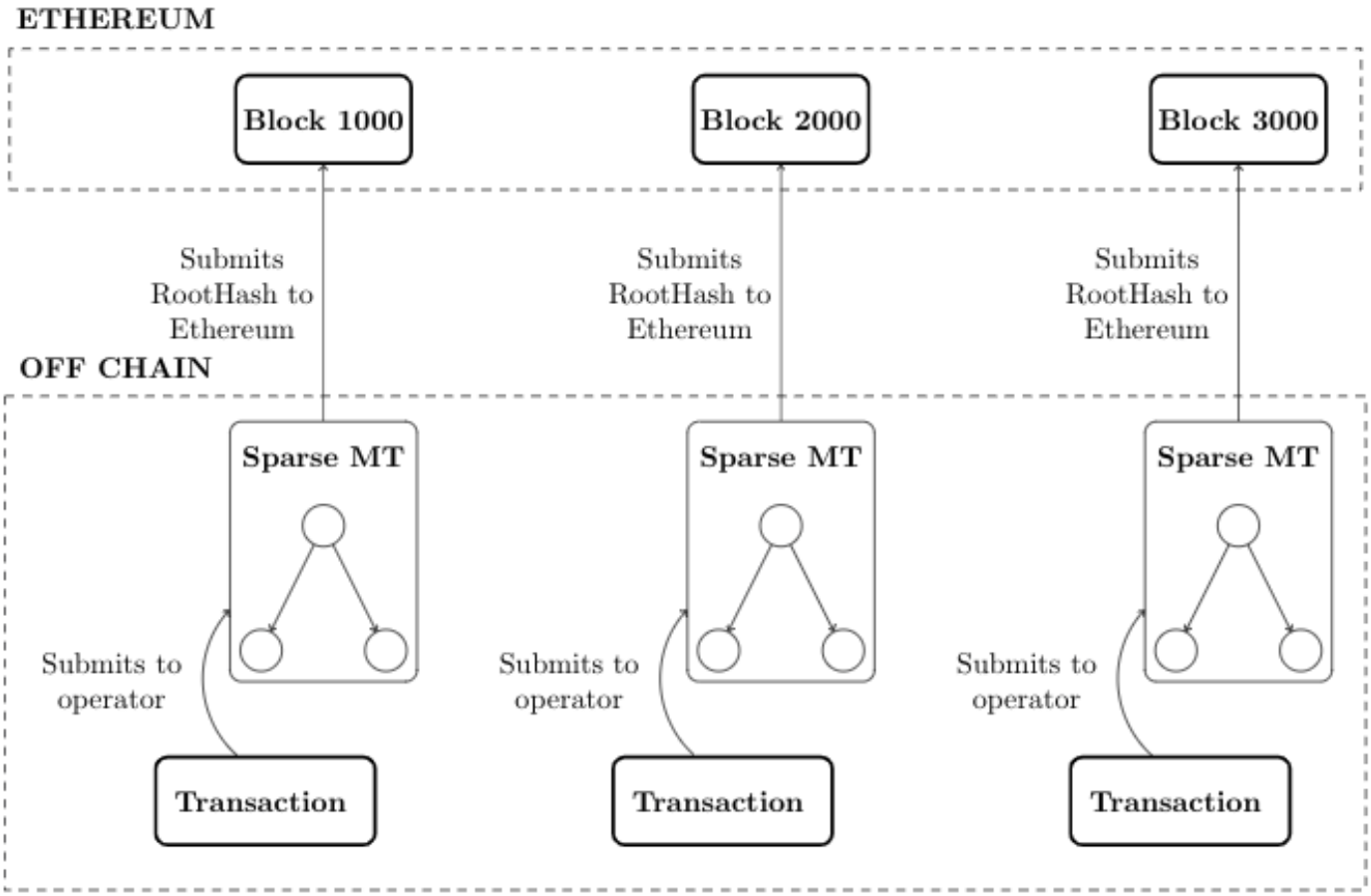

Figure 3.1: SideChain visualization

3.4 Transactions

The main idea behind **Plasma** is to have tokens whose history and true owner can be verified. Since every transaction of a token is backed up by a *RootHash* on **Ethereum**, with the necessary proof, anyone could easily backtrack every single transaction from the token's origin to the last unspent transaction, to determine the owner of that token is. While doing this, invalid transactions can appear, as the transactions are only checked by **the operator** who should be treated as an untrusted party. Nonetheless, thanks to some mechanisms that are explained later, these invalid transactions can be skipped as if they never happened, and are not taken into account in a token's history to determine the owner.

3.4.1 Deposit

Plasma offers a way for users to transfer tokens from one to another for free. However, it has no ability to create tokens. If creation of a token on the **SideChain** was allowed, then **the operator** would have complete control over the generation of these tokens, and the token intrinsic value would decrease. For this, a token will always have to be created on **Ethereum** under the rules of a *ERC-721 Non-Fungible Token*[7] contract. This way, tokens are created in a fair known decentralized way and people can rest assure that no token was created out of this set of rules.

After creating a token on **Ethereum**, the user would have to **deposit** it into the **SideChain**, to be able to use it on **Plasma**. A deposit is basically an *ERC-721* transfer to the **Founding Contract** that will start a chain of events (fig. 3.2):

1. After the transfer, the ownership of the token is passed to the **Founding Contract** and it remains locked while transactions are being generated on the **SideChain**, so that if needed, the true owner can reclaim it at any point. Since the contract has already defined what can and can't do with that token, a mechanism to recover a token is always available (see section 3.5).

2. This **deposit** will create an identifier (known as the **slot**) for this token. This **slot** is a number up to 2^{64} that has to be unique every time a token is deposited.

3. The **Founding Contract** generates a new block on the **RootChain**.The *RootHash* of this block will be represented by the hash of the **slot** and the number of the block will be the following one to the last submitted block. This block will be used as the first *UTXO* transaction of the **slot**. Since these types of transactions are generated *out-of-nowhere*, it makes sense they can only be generated directly in **Ethereum**.

4. When the block is generated on the **RootChain**, an **event** is emitted and listened by **the operator**. The operator now makes notice of this **slot** and a deposit block is saved on the **SideChain** for future validations.

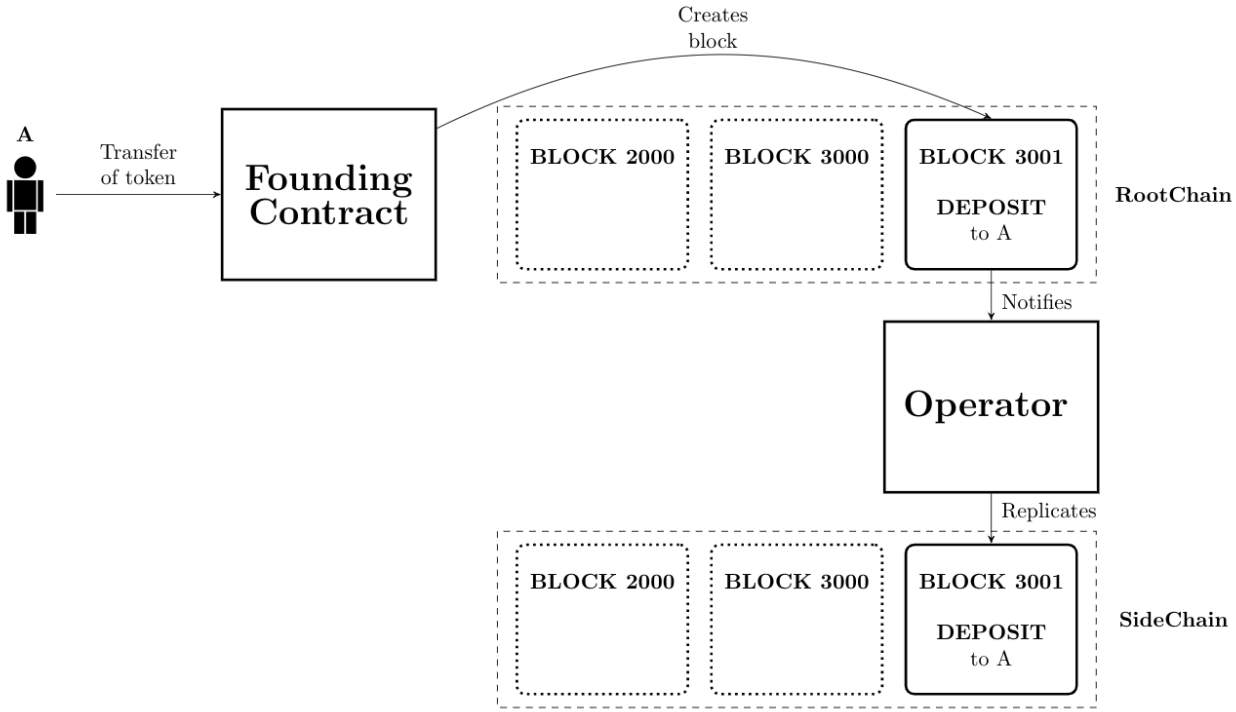

Figure 3.2: Deposit

3.4.2 Transfer

Besides deposit blocks, the rest of the blocks are generated by **the operator**. The *RootHashes* of these blocks are submitted to the **RootChain** to be considered valid. These block numbers increase making steps of 1000, so that every non-deposit block is a multiple of 1000, and any deposit block is not. This makes it easier since whenever a deposit block is generated, it is simply added to the list of blocks, without the fear that another block with the same blockNumber was already created in the **SideChain** due to some race condition.

The main transactions of these blocks are **transfers**. A transfer from **an owner** to **a recipient** of a specific **slot** can be represented by the following structure:

$$\begin{cases} Slot & : \text{The slot number of the token to be transfered} \\ BlockSpent & : \text{The block number of the last unspent transaction, where the owner received the slot} \\ recipient & : \text{The Ethereum address of the recipient of this transfer} \end{cases}$$

A transfer consists of the following events (fig. 3.3):

1. **The owner** signs a message to the operator indicating the attempt of transfer. This message is an *RLP-encoded*[2, Appendix B] array of bytes determined by the following structure:

$$TX = RLP([slot, blockSpent, receiver])$$

2. The operator validates the message received and proceeds to add the transfer to a queue of transactions.

3. When **the operator** decides to submit a block, a set of transactions are selected to be included in it. The *RootHash* of a Sparse Merkle Tree is calculated, where the leaf of the tree are the transactions added to the block.

4. **The operator** creates a block in the **SideChain**, saving its transactions and its *RootHash*. Then submits the *RootHash* with the blockNumber to the **RootChain**

5. The **RootChain** stores this *RootHash* as a new block and now everyone is able to validate all the submitted transactions against it using a *Merkle Proof*, the hash of the transaction and the signature. The transaction's hash is simply the *keecak*[8] output of the transaction's bytes.

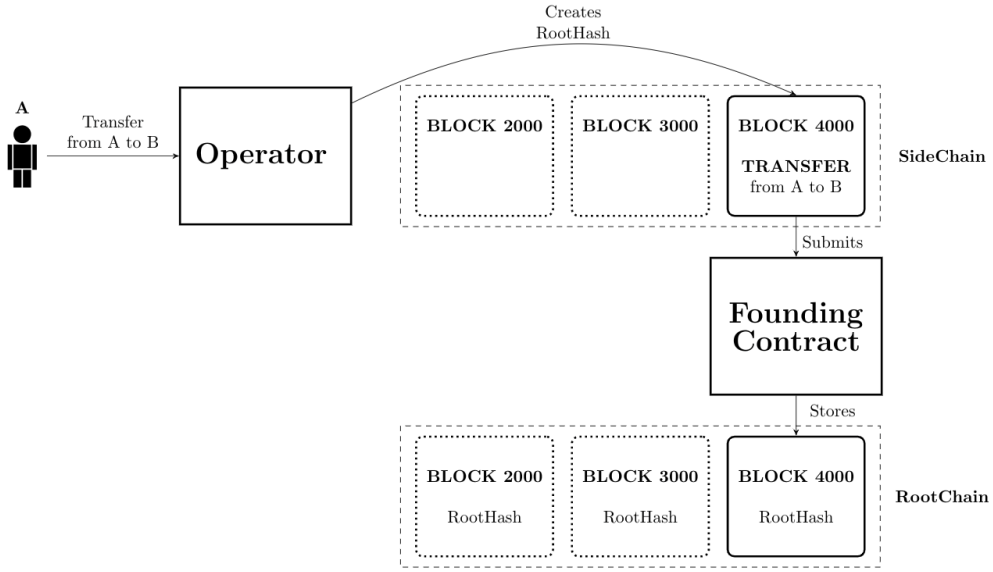

Figure 3.3: Transfer

For a transfer transaction to be valid, the following conditions must occur:

- There must be a valid transaction of the **slot** on **blockSpent**.
- The tx hash must be signed by the receiver of the **slot** in the **blockSpent**.
- The tx must be added in the **slot** position of the **Sparse Merkle Tree**.
- There must not be another valid transaction between **blockSpent** and this transaction's current block for the **slot**.

One key feature is the requirement for a transaction to be included inside the number **slot** leaf of the **Sparse Merkle Tree**. This forces that only one transaction of a **slot** per block can be valid, as two transactions could not share the same leaf. This also allows users to check if a certain block contains no transaction for a certain **slot**. If a **Merkle proof** can be done against *Hash(null)* in a certain **slot**, no transaction spending that **slot** was added to that block. This is useful as one of the requirements for a valid transaction involves having no previous transaction that spends that block. Then, an easy validation could be done by checking the **Merkle Proof** with each block between the *blockSpent* and the current block against *Hash(null)* at that **slot**. With these conditions, and the required proof, any **slot**'s owner can be determined with irrefutable proof.

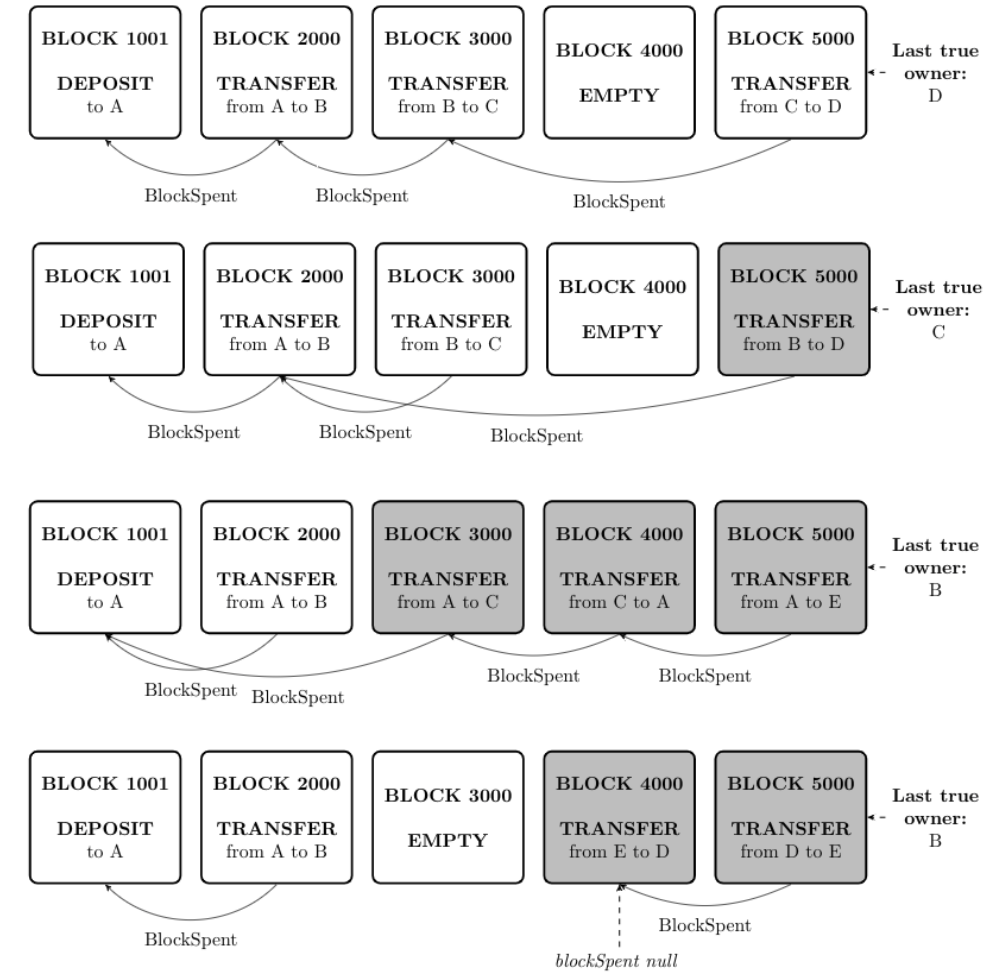

Figure 3.4: Token histories examples

3.5 Exit

Up to this point, is easy to see how **Plasma** can be a great way to economize transaction fees. With grouped and condensed transactions, **the operator** is only charged by submitting one *RootHash* value for any amount of transactions in a block. **The operator** could then monetize in some other way the necessary funds for maintaining this chain, and have transfers be free for every user. However, the token is locked in the **Founding Contract** and that creates a need for a mechanism that allows the true owner to retrieve it. Even if there is a fullproof way of validating a **slot**'s history (fig. 3.4) and true owner, since **the operator** is an untrusted party, there is no guarantee that transfers are not going to be censored by it, rendering

the user's ability to transfer its **slot** invalid. The same problem can be found if **the operator** refuses to provide the necessary proof for a **slot**'s history, as no-one but the true owner can know for sure who the last owner of the **slot** is. For this case, a method to return the token to **Ethereum** when needed is provided.

An **Exit** of a **slot** is the process that a token has to go trough in order to leave the **Plasma** chain and be freed into **Ethereum**. At the end of it, the token corresponding to that **slot** is transferred to the true owner, leaving the lock and becoming available for any other **Ethereum**-related uses. Since the **slot**'s owner could have changed during its lifetime in the **SideChain**, only the latest true owner should be able to exit the token.

For this a **challenge period** is defined, since **Ethereum** has no way of knowing who the true owner is as it only contains *RootHashes*. Making it go through a complete **slot**'s history would defeat the purpose of **Plasma**, for this reason at first, the **RootChain** will believe any user wanting to exit any **slot**. However, during a **7-day period**, any other user has the ability to challenge (all challenge types are detailed in section 3.6) this statement with the necessary proof. If the challenge is successful, the exit is cancelled, if not, the token is released to the user after the period has expired.

The process starts as follows: User **A** is interested in exiting a **slot**. Directly against **Ethereum**, **A** calls one of the following functions on the **Founding Contract** to set that **slot**'s state to exit and start the **challenge period**:

(a) If the user claims no transactions where made on the **SideChain**, then *startDepositExit* should be called. This function will validate that the *msg.sender* of the call is the same as the user that deposited the token in the first place. After that, the **slot**'s state is set to **exiting** and the **challenge period** starts.

```
function startDepositExit(uint64 slot)
    external payable isBonded isState(slot, State.NOT_EXITING);
```

(b) If the user claims at least one transaction was made on the **SideChain**, then *startExit* should be called. This function's parameters, also known as the **ExitData** (fig. 3.5), are a little more complex, and are broken down as follows:

```
function startExit(
      uint64 slot,
      bytes calldata prevTxBytes,
      bytes calldata exitingTxBytes,
      bytes calldata prevTxInclProof,
      bytes calldata exitingTxInclProof,
      bytes calldata signature,
      uint256[2] calldata blocks
   ) external payable isBonded isState(slot, State.NOT_EXITING);
```

- $slot$: The slot to be exited
- $prevTxBytes$: The previous-to-last transaction of the slot
- $exitingTxBytes$: The last transaction of the slot
- $prevTxInclProof$: The previous-to-last transaction's Merkle proof
- $exitingTxInclProof$: The last transaction's Merkle proof
- $signature$: The last transaction's signature
- $blocks$: The number of the 2 last transactions' blocks

With the given proof, the **Founding Contract** is able to check that both transactions were added to the **RootChain** and that the receiver of $prevTx$ signed $exitingTx$. It also does some more basic checks like that the **slot** is the same on both transactions. Since **the operator** can't be trusted, each user should save their **slots**' histories to be able to create an **ExitData** when needed.

Both of these function calls must be accompanied with a **Bond**. This bond is a determined value that will be locked in the **Founding Contract**. If no challenges are generated during the 7-day period, then the bond is returned to the owner with the token. If there is a challenge however, the bond is lost and is credited to the challenger as a bounty. This helps dissuade any fraudulent actor to try to exit

another user's **slot**, since if challenged, the bond will be lost, with nothing gained.

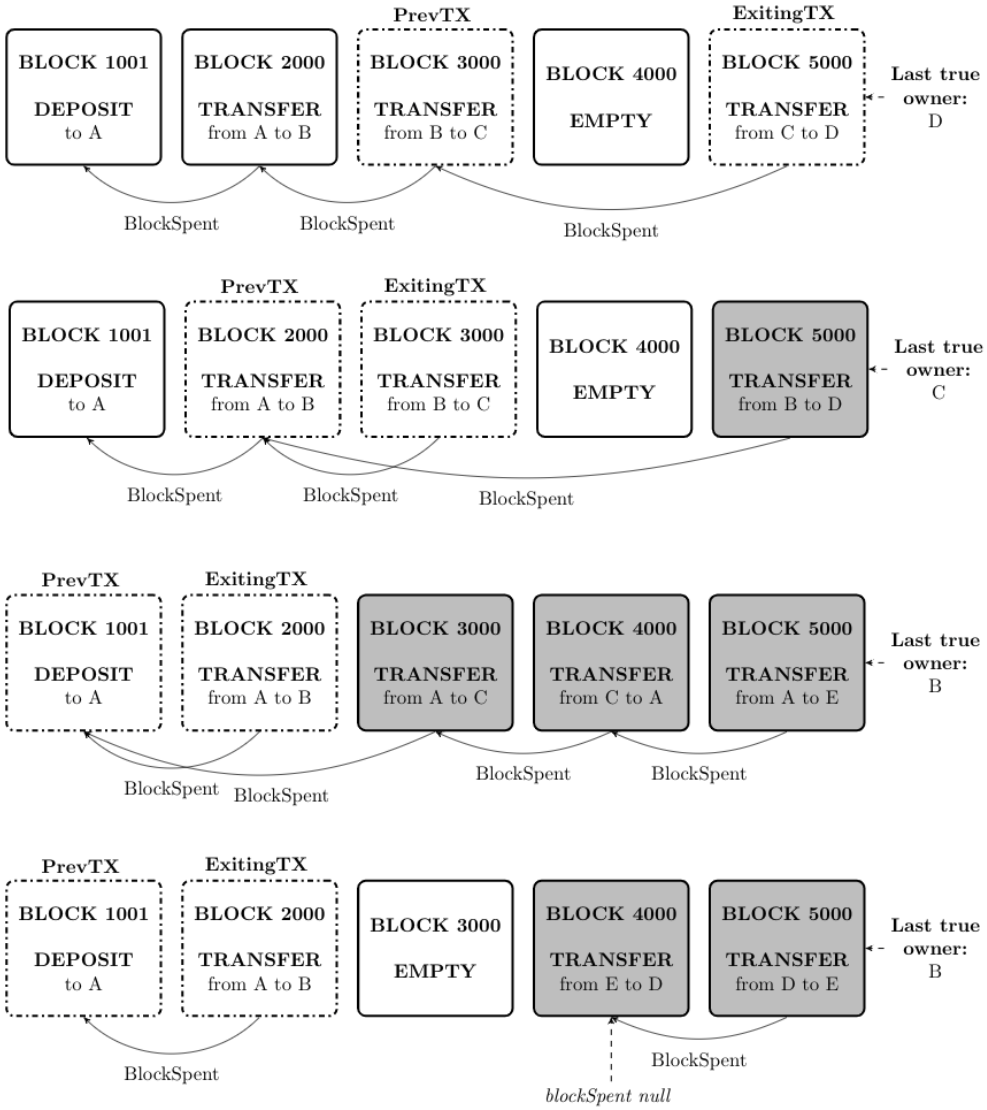

Figure 3.5: Example of ExitDatas for different histories

In summary, to exit a slot a user has to:

1. Use either *startDepositExit* or *startExit*, with its corresponding **ExitData**, and provide a **bond**.

2. The slot is blocked in an **exiting** state until either the **challenge period** ends or a challenge is successfully sumbitted.

3. After the **challenge period** ends, the user will gain ownership of the token in **Ethereum** and the bond will be retrieved. If a successful challenge occur before that time, the exit is cancelled and the bond is given to the challenger.

3.6 Challenges

Challenges are a key factor of **Plasma**. Without challenges, anyone could reclaim any **slot**. A challenge is a proof that the exitor of a **slot** is lying about its ownership and it can be done by anyone wanting to reclaim the **bond**. If the exitor is the true last owner and the **ExitData** is correct, no challenge will succeed against it.

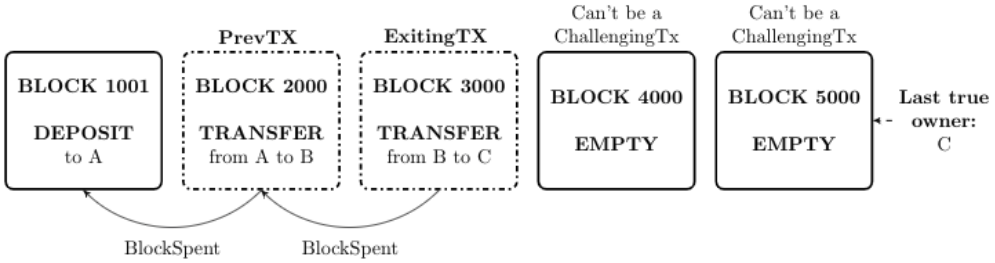

Figure 3.6: Unchallengeable exit

However, under any fraudulent scenario, there is at least one challenge that will succeed. The challenges can be classified as follows:

3.6.1 Challenge After

A **Challenge After** prevents a user from trying to exit an already spent transaction (fig. 3.7).

```
function challengeAfter(
    uint64 slot,
    bytes calldata txBytes,
    bytes calldata proof,
    bytes calldata signature,
    uint256 blockNumber
) external isState(slot, State.EXITING);
```

$$\begin{cases} slot & \text{: The slot being exited} \\ txBytes & \text{: The transaction spending the exiting transaction} \\ proof & \text{: The transaction's Merkle proof} \\ signature & \text{: The transaction's signature} \\ blockNumber & \text{: The transaction's block} \end{cases}$$

If a transaction that spends the **exiting block** exists, anyone can provide it to the **Founding Contract** and reclaim the **bond**, canceling the exit.

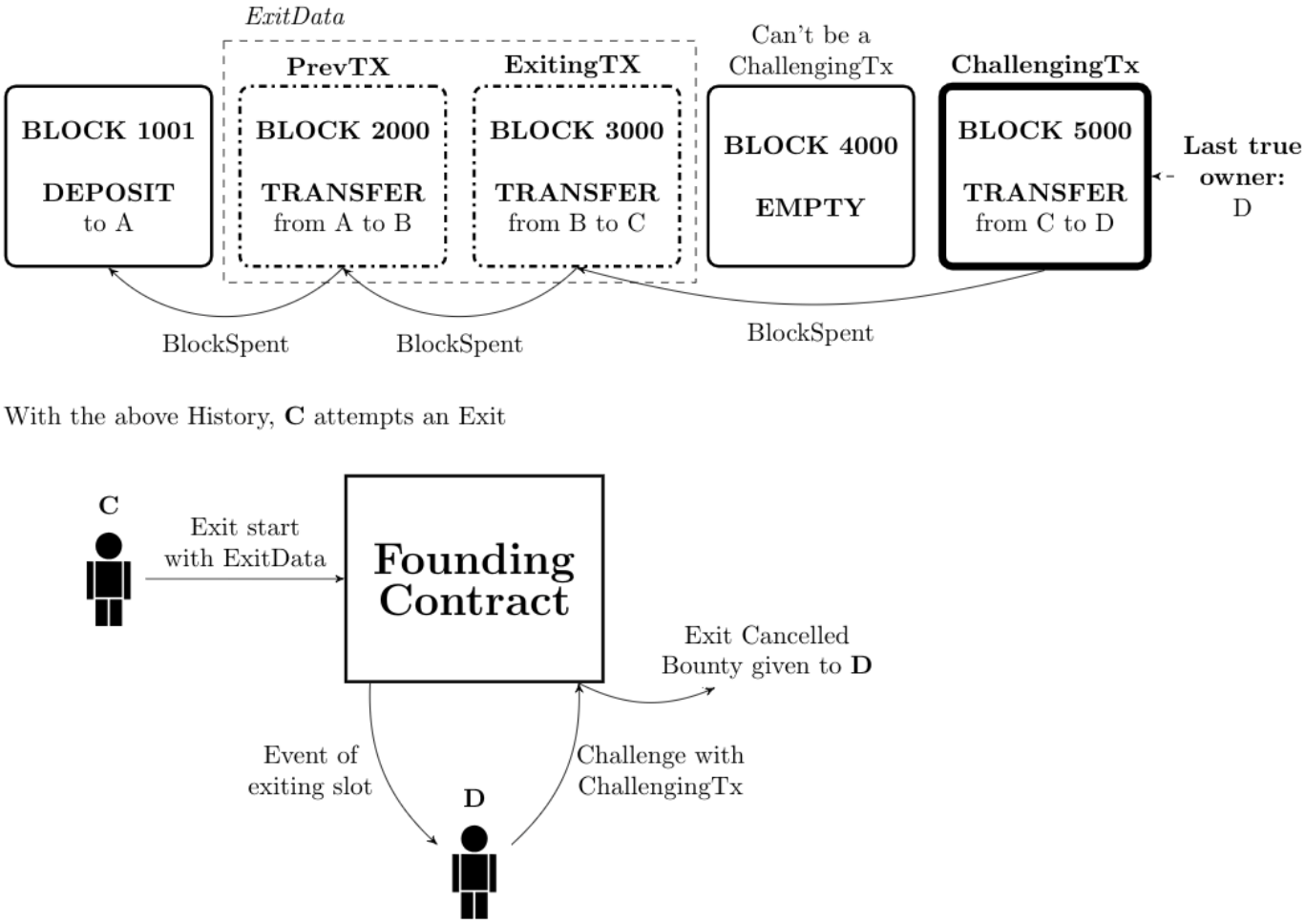

Figure 3.7: Challenge After

3.6.2 Challenge Between

A **Challenge Between** prevents a user from trying to exit a double spend (fig. 3.8). If a transaction that happened before the exiting transaction exists, and it spends the prevTransaction's block, then **the operator** allowed a double spend. This double spend means the **exitingTx** is invalid, and the exit is canceled, providing **the bond** to the challenger.

```
function challengeBetween(
   uint64 slot,
   bytes calldata txBytes,
   bytes calldata proof,
   bytes calldata signature,
   uint256 blockNumber
) external isState(slot, State.EXITING);
```

- *slot* : The slot being exited
- *txBytes* : The transaction spending the prevTransaction
- *proof* : The transaction's Merkle proof
- *signature* : The transaction's signature
- *blockNumber* : The transaction's block

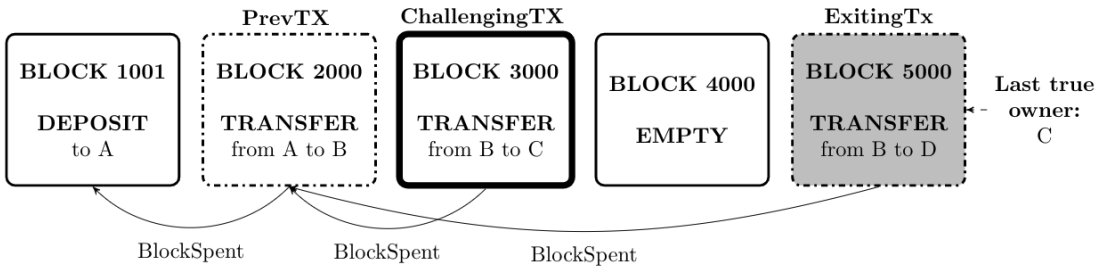

Figure 3.8: Challenge Between

3.6.3 Challenge Before

A **Challenge before** is the most complex challenge as it requires interaction between the exitor and the challenger. To begin, the challenger exhibits an old transaction, claiming this to be the true last transaciton (for example, in fig. 3.9, the transfer in from D to E in block 5000). This exhibit is also **bonded**, to dissuade fraudulent actors to attempt erroneous challenges. If the exitor is not able to provide a transaction that spends this exhibited transaction in the 7-day period, then there is an irregularity in the **slot**'s history, and the exitor is not the true owner of the **slot**. The exit is cancelled after the period ends and the challenger is rewarded with the exit **bond**.

```
function challengeBefore(
      uint64 slot,
      bytes calldata txBytes,
      bytes calldata proof,
      uint256 blockNumber
   ) external payable isBonded isState(slot, State.EXITING);
```

- *slot* : The slot being exited
- *txBytes* : The tx claimed to be the slot's last tx, previous to the exitingTx
- *proof* : The transaction's Merkle proof
- *blockNumber* : The transaction's block

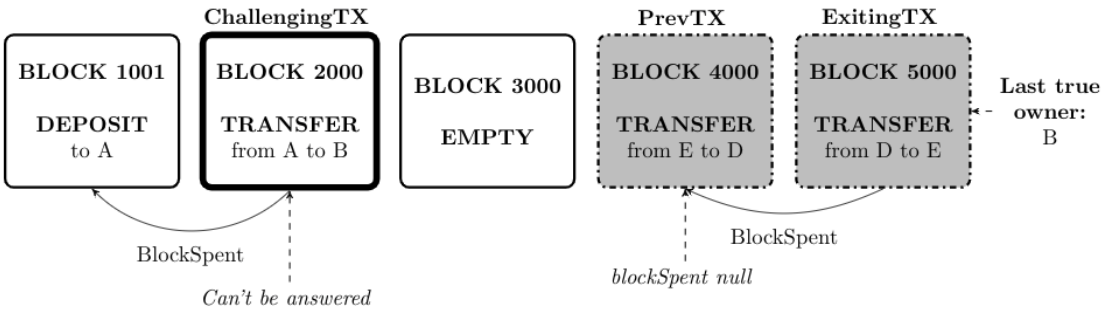

Figure 3.9: Unanswerable Challenge Before

However, if the challenge is issued with a valid transaction in the **slot**'s history,

it can be answered so that the exit continues and the **challenge bond** is credited to the responder (fig. 3.10).

```
function respondChallengeBefore(
      uint64 slot,
      bytes32 challengingTxHash,
      uint256 respondingBlockNumber,
      bytes calldata respondingTransaction,
      bytes calldata proof,
      bytes calldata signature
   ) external;
```

Many challenges before can be queued against an **exit**, and they are removed from the queue as are answered. If at least one of them stays when the window ends, then the oldest challenge is the one credited with the **exiting bond**. This prevents someone from making an unlawful *challenge before* and answering it just as the **challenge window** finishes. If all the challenges were answered, then the **exit** is finalized as usual and the token is released.

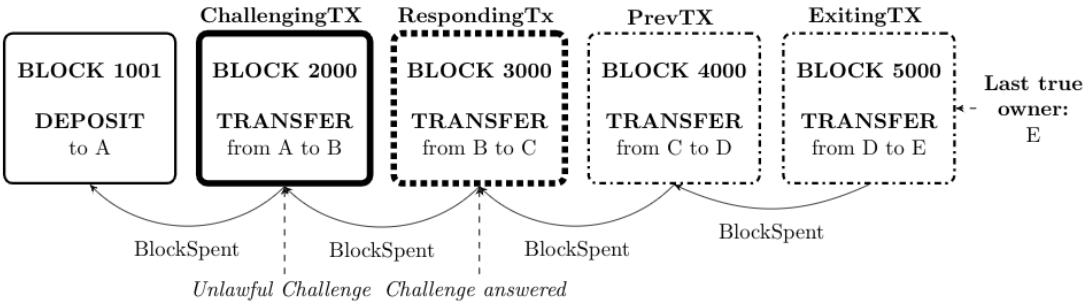

Figure 3.10: Answered Challenge Before

3.7 Considerations and limitations

Plasma is under no conditions a full-proof solution for **Ethereum** and it has its clear downsides. **Plasma** works under the presumption that a check against the **RootChain** is done at least once a week for unlawful exits. Although **the operator**

is the most interested party in having the **SideChain** work as intended and also has the full history of every **slot**, then it will probably be the first challenger to any unlawful exits.

Some issues start appearing as the trust to **the operator** diminishes. Even though a token will never be able to be stolen, for this to be avoided, the user with the token must be connected to **Ethereum** at least once a week. Moreover, the user must store all the **slot**'s history to be able to challenge an exit. These two tasks are not easy to achieve for day-to-day users, however, the fact that it can be done is a big deterrant for anyone trying to cheat the system.

Another issue is regarding the cost of depositing and exiting a token. A deposit's cost is a simple transaction, however, an exit's cost is not only the computational issue, but also the **bond** a user has to freeze for a week and the time the token has to wait until the user can recover it. If **the operator** misbehaves allowing an invalid transaction to go through, the user has to exit the **slot** or else that transaction can then be used as a **challenge before** in the future. There are some proposals on how to punish **the operator** if an invalid transaction is committed, however due to scope issues, they were not investigated in this project.

This comes to the realization that, if **the operator** misbehaves in some way, then some of the incentives to have tokens deposited on the plasma chain are lost, and a mass exit of the **slots** for users may start. As a starting point for a scalability solution is a good step in the right direction, but there is still a long way until it becomes more robust.

Atomic Swaps

4.1 Introduction

Once a **Plasma Chain** is working and being used, one of the key features missing is the ability to swap **slots** between users. Transferring **slots** is great as a starting point, however when expecting something in return, that can't be validated by the transaction under these initial conditions. This limits the situations in which a **slot** transfer could be used, as there must be some other mechanism outside the side-chain to validate an equal exchange of goods or services. This is no problem in blockchains such as **Bitcoin**, since money exchange is viewed as a *"pay first - receive good and services after"* transaction in the world. But if a market place of sorts is to be implemented, there has to be a way to validate both ends of the agreement.

4.2 Atomic Swap Transaction

The basic idea of an atomic swap is a set of transactions between **A** and **B** where **A**'s **slot** must change ownership to **B** if, and only if, **B**'s **slot** changes ownership to **A**. For retro-compatibility and simplification purposes, the new swap transactions should be compatible with previous definitions of **Plasma**'s transactions, only varying in its encoding and validation. This means the *ExitData* can be viewed as **TxBytes**, **ProofBytes** and **Signature**. Moreover, it must maintain the core attributes of a **Plasma** transaction: They must both spend a previous block and have the ability be spent by a future block.

4.3 The branching history dilema

One of the key aspects of **Plasma** is **history validation**. Before receiving a **slot**, to consider it as a valid transaction, the user must first validate the true ownership of that **slot**. The history must be validated as explained before: The inclusion and omission of transactions for each block must be checked for that **slot** to be considered valid. This task, while still big enough for day-to-day purposes, remains **O(n)** time complexity regarding the blocks mined since the token's deposit. For **Plasma** to be feasible, this **O(n)** complexity can't increase.

However, taking a look at the initial definition, an atomic swap should be rendered invalid if the ownership of the **slots** did not change. Let say **A** is swapping its *token 1* with **B**'s *token 2*, however, **B** is not the true owner of *token 2*, as it was forged by **the operator** or a transaction spending *token 2* was already committed. Then, if a swap is created, that swap is considered invalid, as the true owner of *token 2* can challenge the exit. This case is illustrated in figure 4.1, where **B** swaps token 2 with token 3 supposedly owned by **C**, but really owned by **D**. If **B** tries to exit to token 3, then D would be able to challenge it.

Following the previous example, if now **B** tries to transfer *token 1* (which is an invalid transfer since the swap was invalidated), the receiver must now check *token 1*'s history. *Token 1*'s history appears to be normal, however, if the receiver proceeds to check whether the swap was valid or not, he now must check all of *token 2*'s history as well, determining now that *token 2*'s history is invalid, thus the swap was invalidated, thus **B** is not the owner of *token 1*.

It is easy to see then how this **history validation** can fall under a $\mathbf{O}(2^n)$ complexity as swaps generate branching factors in which the receiver must also verify the swapping **slot**. If those **slots** had swaps, it has to also verify those swapping **slots**, so long and so forth. This is not acceptable for **Plasma** since the **history validation** is one of the founding blocks of the concept, and keeping it under a reasonable computable time is crucial for its success.

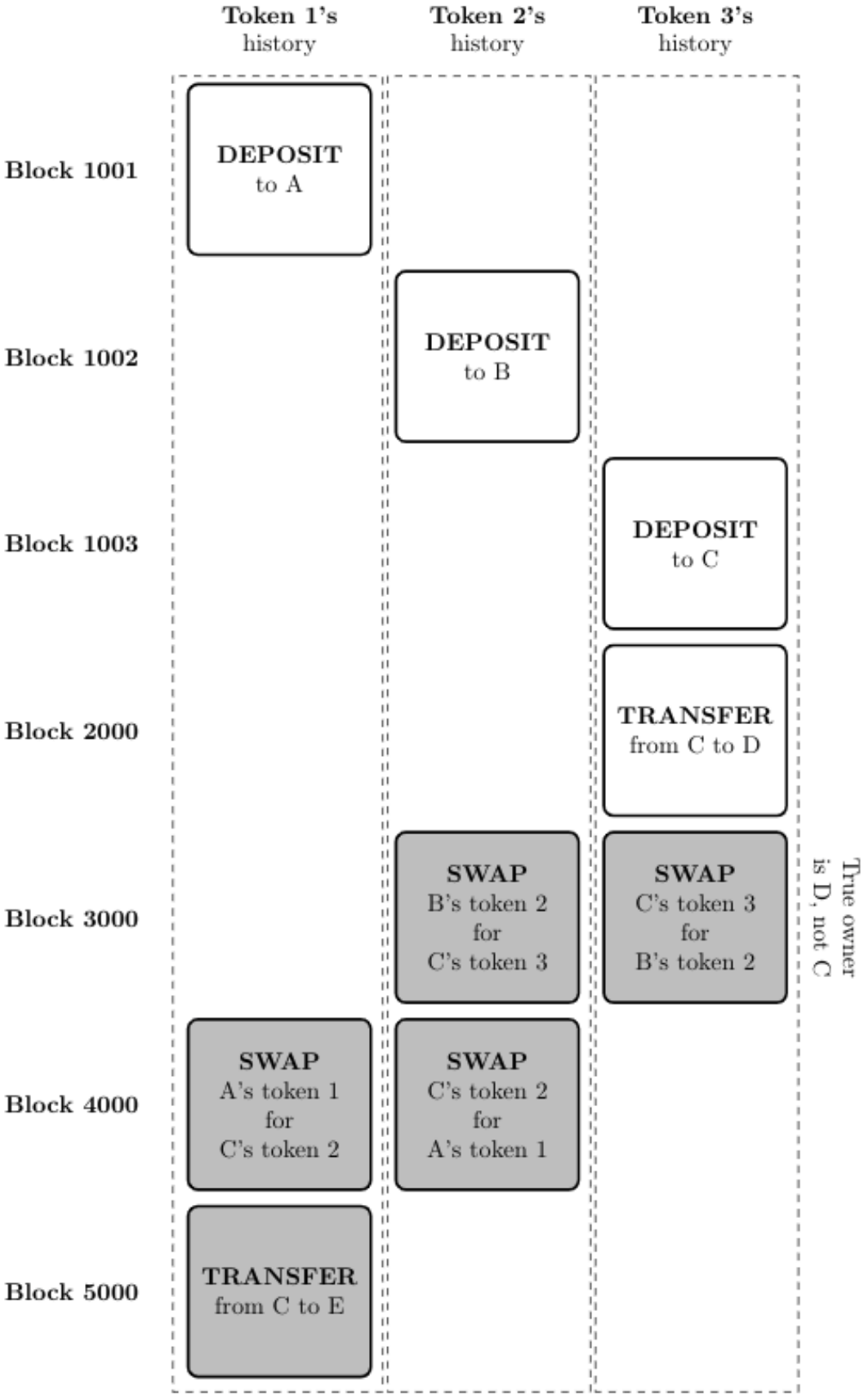

Figure 4.1: Conflicts in branching history

4.4 Secret Revealing Swaps

The next logical step is to rely the validation of every swap to the interested parties. If one of the interested users fails to validate a history before using a swap, is no one but that user's fault, and the swap should be valid for one of the parts. This completely removes the issue with branching history, as when validating a swap, the other **slot**'s history is not taken into account for that **slot**'s validity. This way if **A** and **B** swap *token 1* and *token 2* respectively, and *token 2* is forged, since **A** accepted the swap without validating *token 2*'s history, **A** now looses the ownership of *token 1* while not getting anything in return, since the true owner of *token 2* can always challenge an exit.

However, under these conditions, there is a situation in which a user can validate a history, accept the swap, and still find itself under the loosing end of the agreement (fig. 4.2). Continuing with the example, **A**'s *token 1* is being traded with **B**'s *token 2*. **A** validates *token 2*'s history and up to that time, **B** is the true owner of it. **A** and **B** proceed to sign a swap and send it to **the operator**. Before including the swap to the chain, **the operator**, after it has received the swap, decides to commit a transfer of *token 2* from **B** to **C**, signed by **B**. If this transfer is committed before the swap, then **C** is the new owner of *token 2* and the *token 2*'s part of the swap is invalid, since **B** is no longer the true owner when the swap is submitted. However, *token 1*'s part of the swap is still valid, thus resolving in **C** keeping *token 2* and **B** receiving *token 1*, all due to timing decided at the end on the operator, with no way for **A** to prevent this from happening.

To avoid this, and keep the lineal history, a commit-reveal mechanism is proposed[9]. Cryptography has been using commit-reveal mechanism for a long time to get around this kind of situations. The basic idea of it is: The user A generates a random secret, keeping it safe from being revealed. Provides the interested party of the commit the hash output of that secret. After the commit is accepted by every party, the secret can be revealed to seal the deal, if the secret is not revealed, the transactions is invalid.

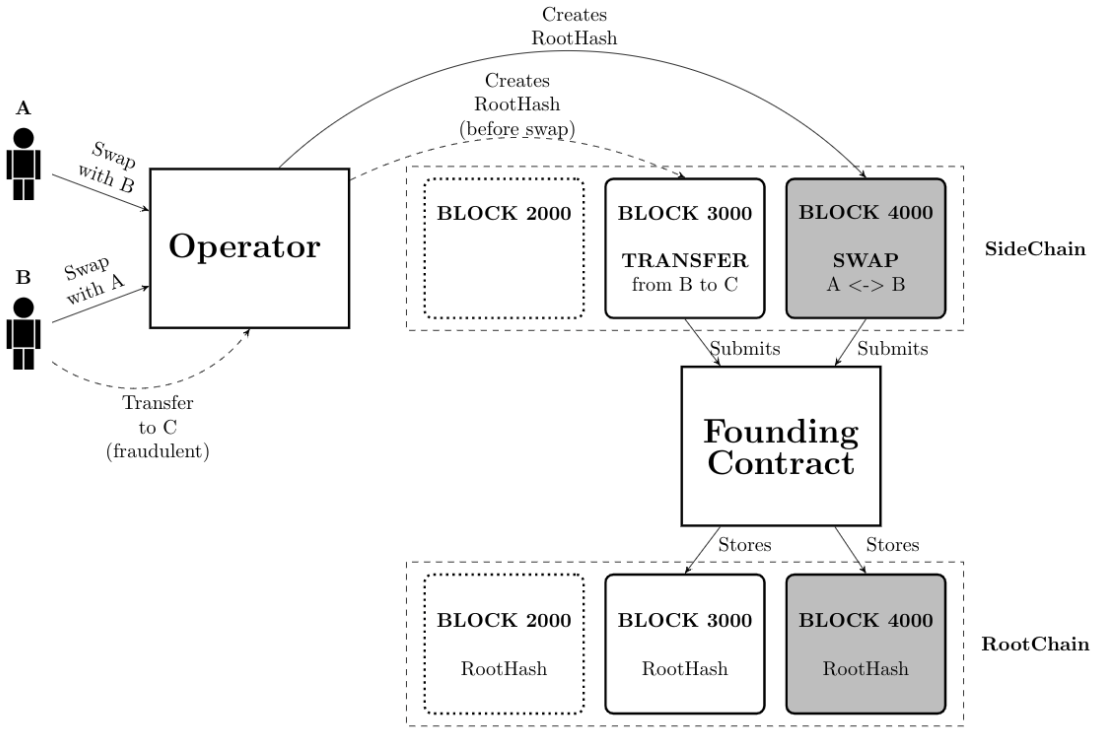

Figure 4.2: Operator misbehaving on a swap

This concept can be tied to a swap transaction to avoid the issue with history changing in between the submission to **the operator** and the commit it makes to the *RootChain*. With this, a secret can be embedded into the swap until it is committed to the side-chain. Once committed, the users are free to then validate the other party's **slot up until the committed block**. If the history validation is correct, then the secret can be revealed. The swap is only considered valid if both secrets are revealed withing a day of the submission of the block. If an alteration to the history is tried to be sneaked in before the swap is committed the user will notice it, thus never revealing the secret, and letting the swap expire after a day.

If the user fails to validate the history after the swap was committed, and still reveals the secret, then his token is lost with nothing in return, as the true owner of the **slot** that the user received can challenge an exit, but this is only because of the user's fault. This method maintains the history lineal as a token's validity only depends on its previous transactions.

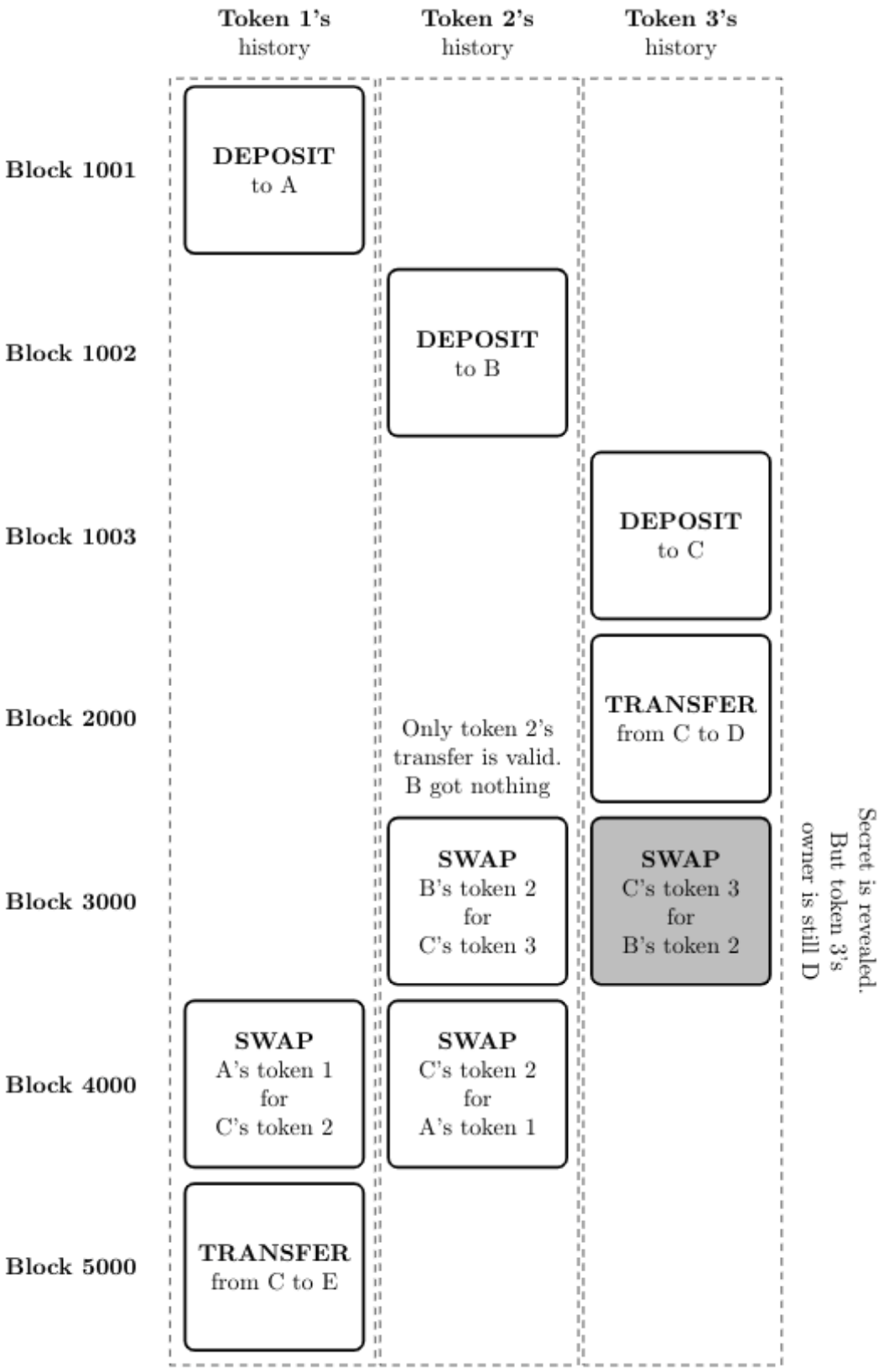

Figure 4.3: Revealing a secret on fraudulent swap

4.5 Secret Revealing Chain

The **Secret Revealing Chain** is a collection of blocks, just like the *RootHashes* of the **RootChain** that are also saved in the **Founding Contract**. Unlike the **RootChain**, this new chain has no restrictions on the order of the submissions or the interval between them. The only restriction is that a block with the same *blockNumber* must have been submitted to the **RootChain** within the day for the submission. The contents of this chain are also *RootHashes* of Spare Merkle Trees, however unlike the **RootChain**, the leafs of these trees are not transactions, but rather secrets.

```
Interface FoundingContract {
   SecretRevealingRootChain::Map<BlockNumber, RootHash>
   function submitSecretRevealingBlock(BlockNumber, RootHash) onlyOwner
}

Interface Operator {
   SecretRevealingSideChain::Map<BlockNumber, (slot, secret)[]>
   function swap(swapTxBytes, signature) (section 4.6)
   function revealSecret(slot, BlockNumber, secret)
}

Interface SwapProof: Proof {
   inTxBytes (section 4.6)
   signature,
   AtomicProof, (section 4.7)
   SecretA,
   SecretB
}
```

For every swap of **slot N** and **slot M** in the **block B** in the **RootChain**, there should be one secret in the **Nth** position and another on the **Mth** position of the Sparse Merkle Tree whose *RootHash* is submitted on the **Bth Secret Revealing Block**. These secrets must match, after hashing them, with the secret hashes

submitted in the swap. This way, whenever any amount of swaps are condensed into one *RootHash*, the same amount of secrets are condensed into another *RootHash*, preventing **the operator** from having to submit multiple secrets at a time.

This block is limited to a day after the original **RootChain** block was submitted so in case a user wants to cancel a swap, it will expire after a day. Otherwise, there would be no way to be sure if a swap is invalid or will become valid when reveling a token history.

4.6 Implementation

Let **A** and **B** be the parties interested in the swap of their respectives **slots** located in **SlotA** and **SlotB**.

Let **SecretA** and **SecretB** be a random 32-bytes number for each respective party, secured from being discovered until revealed.

Let **HashSA** and **HashSB** defined as:

$$HashSA = Keccak256(SecretA)$$

$$HashSB = Keccak256(SecretB)$$

Let **TxA** and **TxB** be the transaction components for each party defined as:

$$TxA = RLP([slotA, blockspentA, HashSA, B, slotB])$$

$$TxB = RLP([slotB, blockspentB, HashSB, A, slotA])$$

Let **HashA** and **HashB** be the *keecak* output of these bytes.

let **SignA** and **SignB** the signature proof of each party defined as:

$$SignA = signECDSA(HashA, PkA)$$

$$SignB = signECDSA(HashB, PkB)$$

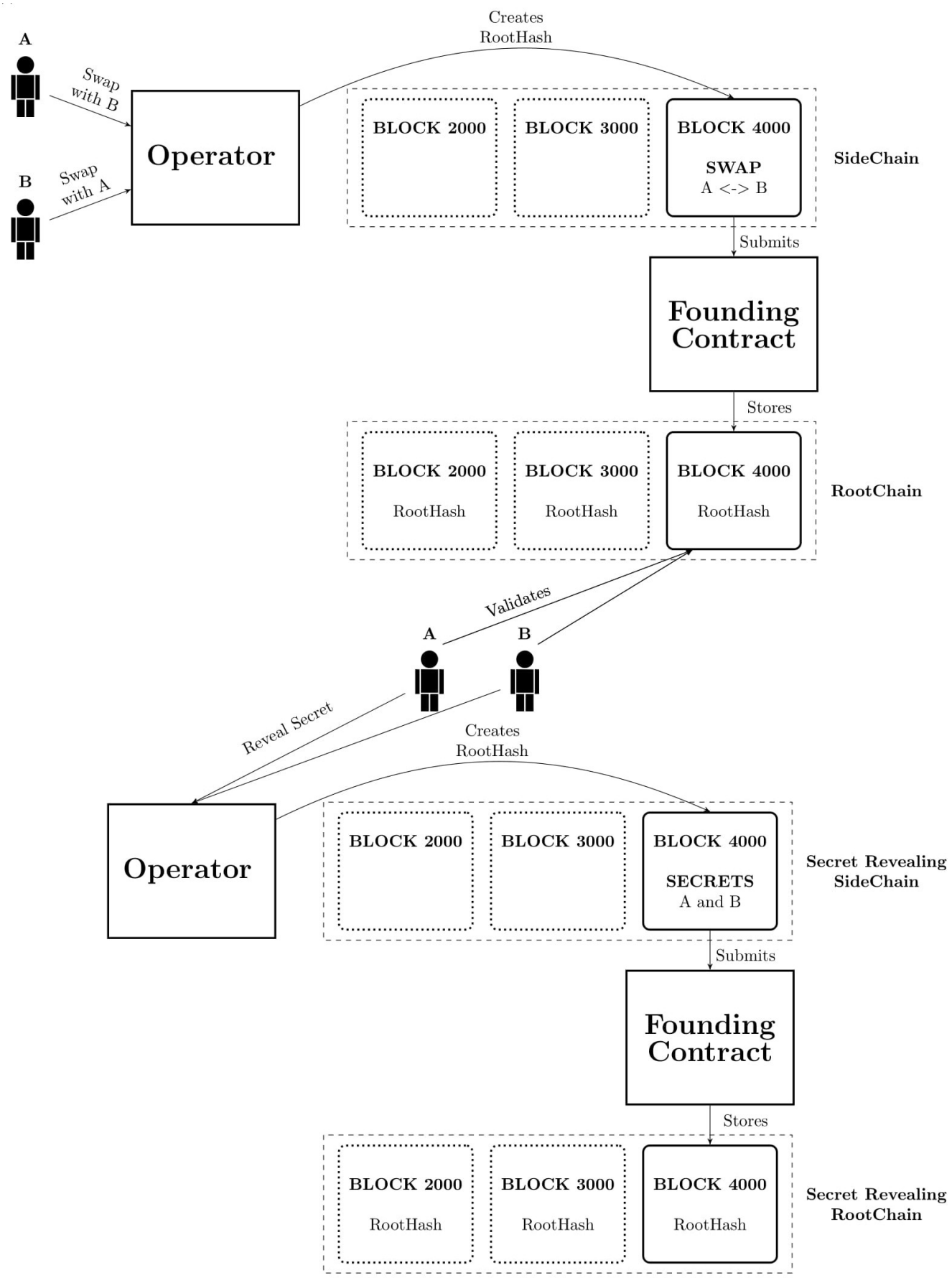

Figure 4.4: Atomic Swap

As show in the figure 4.4, the process to swap two tokens between **A** and **B** is the following:

1. Both parties send to **the operator** the transaction component and the signature announcing an intent of swap. **The operator** must verify their correctness, and wait for when it possesses both swap components. If both transactions are not submitted in a reasonable time frame, **the operator** is free to remove them from the queue.
2. When both of the transactions are received, then their hashes are added to the **Sparse Merkle Tree** of a block, each hash in their corresponding slot. The *RootHash* of the block is then submitted to the **RootChain**.
3. There is now an **intermediate proof** that **the operator** must provide to the users for them to check that both hashes are added to the **RootChain** correctly. Also the **intermediate txBytes (inTxBytes)** have to be provided so the users can check the opposing hash was generated correctly. Any party will only need one of the intermediate txBytes to be able to validate the counterpart against the **RootChain**.

$$inTxBytesA = RLP([slotA, blockspentA, HashSA, B, slotB, blockspentB, HashSB, A, signB])$$
$$inTxBytesB = RLP([slotB, blockspentB, HashSB, A, slotA, blockspentA, HashSA, B, signA])$$

4. Once both transactions are verified by the respective parts, then the parts proceed to reveal the secret to the operator. The operator has a 1 day period to submit a **SecretRevealing RootHash** for the corresponding block. After that, the contract will not accept any new SecretRevealing RootHash for that block and the swap will be invalid. Once the SecretRevealing RootHash is submitted, the transaction is considered valid.

   A **SecretRevealing RootHash** of block X is the *RootHash* of a Sparse Merkle Tree (SMT) with the same depth as the original SMT, but where the leaf S correspond to the secretS of the slotS in an atomic swap added in the block X. This tree will be called **SecretRevealingMerkleTree of Block X (SRMT(X))** (fig. 4.5).

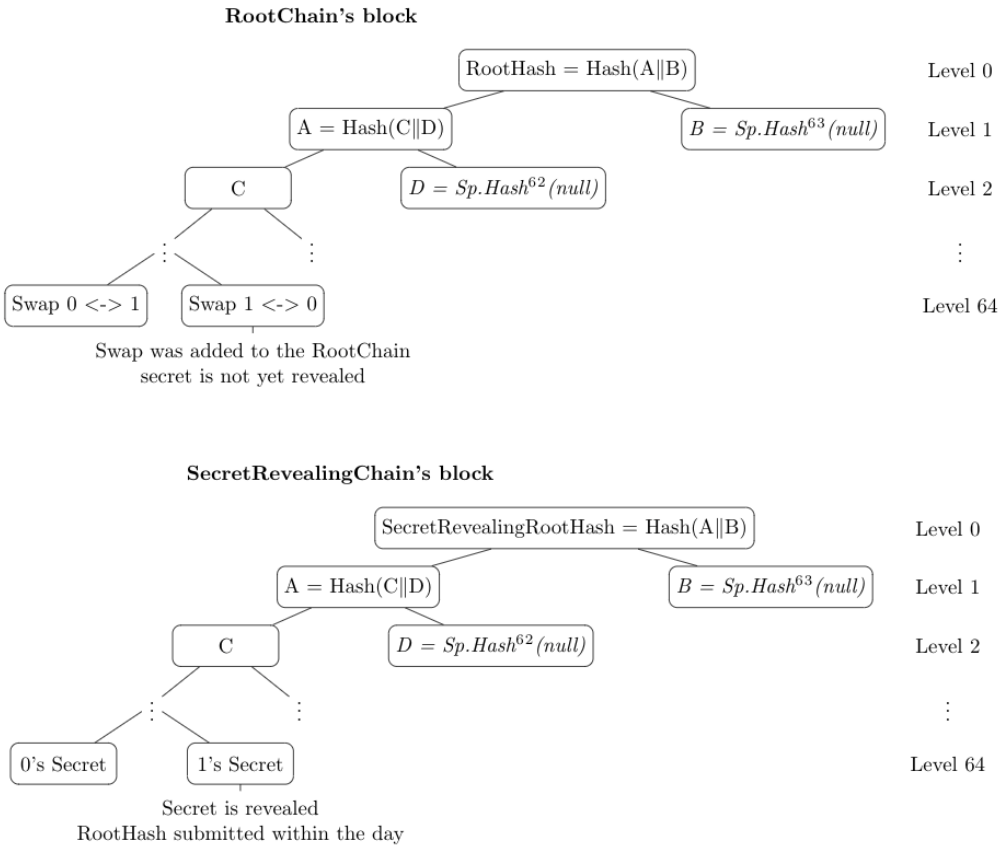

Figure 4.5: Creation of SecretRevealingRootHash

SecretRevealing RootHash of Block X = RootHash(SRMT(X))

SRTM(X) = Sparse Merkle Tree with each leaf S:

$$LeafS = \begin{cases} Hash(null) & \text{Leaf S of SMT of Block X is not an Atomic-Swap} \\ SecretS & \text{Leaf S of SMT of Block X is an Atomic-Swap} \end{cases}$$

4.7 Validation

In order for an atomic swap be validated, the following values must be provided:

$$TxBytesA = RLP([slotA, blockspentA, SecretA, B, slotB, blockspentB, SecretB, A, SignB])$$

$$AtomicProofA = RLP([proofInclA, proofInclB, proofInclSecretA, proofInclSecretB])$$

$$SignA = ECDSASign(HashA, PkA)$$

$$BlockNumber = \text{BlockNumber which included the swap}$$

With these values, the following criteria must be met:

- The **SecretRevealing RootHash** has been commited
- $HashSA = Keecak(SecretA)$
- $HashSB = Keecak(SecretB)$
- $TxHashA = Keccak256(slotA, blockspentA, HashSA, B, slotB)$
- $TxHashB = Keccak256(slotB, blockspentB, HashSB, A, slotA)$
- $SignA$ validates for TxHashA, A
- $SignB$ validates for TxHashB, B
- An inclusion proof is done with TxHashA and proofInclnA in the rootHash of the SMT of blockNumber for the leaf SlotA
- An inclusion proof is done with TxHashB and proofInclB in the rootHash of the SMT of blockNumber for the leaf SlotB
- An inclusion proof is done with SecretA and proofInclSecretA in the rootHash of the SRMT(blockNumber) for the leaf SlotA
- An inclusion proof is done with SecretB and proofInclSecretB in the rootHash of the SRMT(blockNumber) for the leaf SlotB

4.8 Attempts of attack

The following cases are taken into account using this method.

- If **the operator** hoards the transaction for future use, just as any other plasma transaction, an exit can be performed. Also any new transaction that also spends the *blockSpent* renders this transaction invalid.
- If **the operator** does not include one of the 2 parts of the swap, the transaction is invalid.

- If **the operator** submits only one secret, the transaction is invalid for both parties.
- If one of the coins is spent before the swap is committed, the parties can validate this and not reveal the secret, invalidating both parts of the transaction 1 day later.
- If **the operator** does not provide the necessary proof to validate the coins, the parties never reveal the secret, invalidating the transaction 1 day later.

Force Move Channels

5.1 Definitions

5.1.1 Turn-Based game

A turn-based game is defined as any game in which the players take turns making decisions that modify in some way the state of the game. For this project, the focus will be limited to any 2-player game in which the decisions made are selected from a closed set of decisions. Many games of this sort are required for both players to make the decision simultaneously, such as Rock-Paper-Scissor. For this, an easy work around will be explained later.

5.1.2 State-Transitioning game

A state-transitioning game is defined as any game in which at any point can be described as a serializable state. This state serves as stand alone starting point to, give the decisions, transition to the next state. There must exist 3 deterministic functions regarding this serialized state. **IsStartStateValid** will determine whether a state could be considered a starting point for a game. **IsValidTransition** will determine the validity of a transition from a state A to a state B. **IsEndState** will determine whether a state could be considered as the final state and the game is considered concluded. It should be noted that given these restrictions, any games that require a look-back functionality of more than 1 state to determine a transition can not be valid.

5.2 Game Channel

5.2.1 Basics

In order to allow users to have a continuous experience when playing, transactions should be quicker and cheaper, even free, as a great deterrent from blockchain gaming is confirmation times and transactions fees. A way to overcome these problems are **Game Channels**. A Game channel is a scalability solution in which a PVP (Player vs Player) computation is done outside of the blockchain while at any moment having the accountability for it. Similar to other scalability solutions for offline computing, for this method to work a Game Channel consists of 3 basic components.

A **founding transaction** in which both (or more) parties agree on the stakes in play, the game conditions, rules, initial state and more. Any stakes at play must be locked in a contract in which both parties authorization is needed in order to release it. A **state transition** mechanism in which the parties agree on how states must be handled and signed to one another in order to proceed with the game. **An exiting condition** in which the winning party could prove, giving a signed state, that both parties have reached an ending state and thus reclaim the earnings, without the need of all the intermediate computation. Both the founding transactions and the exiting condition must be done in the blockchain, as are needed to handle the assets, while the intermediate state transitions can be handled offline between the interested parties.

5.2.2 Force Move Channel

Diferentiating from other scalability solutions, where any current state can considered as final and can be exited, some games can't be exited in intermediate states as there is no way to determine the winner of the exchange during these. As such, a mechanism in which a player can't disappear from a game when the odds are not in its favor is needed in order to avoid these behaviours. For that, the only valid intermediator is the Blockchain itself, in which any player can **request a move** from the other player within a **challenging window** time. If a player fails to respond,

the win is conceded to the player who made the challenge.

As there is no way to determine if a player is doing this challenge because the opponent failed to answer or because they fail to accept a valid answer, there is no way to determine who should be punished in this situation, because of that no punishment could be applied other than the transaction fees required for both players to require and answer the challenge.

5.2.3 Integration with Plasma

In order to validate the idea of **Plasma** as a scalability solution to be implemented for blockchain gaming, more than mere trading should be possible to be done when locking tokens on the side-chain. For this to work, any owner of a plasma token should, by the same process in which an **exit** is created, claim to own a token when creating the channel, and having the challenge period just the same way an exiting a token has. Since the downside of not challenging a token is less severe (no token can be stolen this way) the challenge periods can be shortened in order to maintain the streamlined aspect of a match.

The basic idea is to generate **exit datas**, without the actual exit, for every token being used in the channel. Each of these **exit datas** can be challenged independently the same way a plasma exit can be challenged. If the challenge is successful, then the stakes are given to the challenger as a reward.

5.3 Implementation

5.3.1 Channel Manager

One of the most expensive parts of creating a channel is deploying the **Fundation Contract**. In this case, the user may not only pay the fees of the creation of a contract but at the same time have access to the code that will be used. Also both parties must have that code validated and audited. These costs can be diminished

when there is an interested party in having others use these channels, as is the case of **an operator** with its game. Having a main contract do the parts of a **Channel Manager** for multiple channels means less fees, more trust on the code and an easier onboarding process. This way, a structure can be created to represent a channel, and have all these channels sharing the same contract code. Moreover, this channel manager allows users to watch their tokens being used by another players in a centralized address and challenge them when that happens.

5.3.2 Channel Structure

A channel can be defined as a data structure composed of:

- **channelId** which should be unique.
- **ChannelType** which determines the address of the game rules.
- **FundedTimestamp** which determines when the channel was funded.
- **Stake** which determines the bet amount.
- **Players** that determine the interested address on the channel.
- **PublicKeys** that determine the keys to be used to sign the states.
- **InitialArgumentsHash** used as reference when submitting an initial state.
- **ChannelState** representing the stage of the channel (fig. 5.1):
  - **INITIATED:** A player proposed a game, locking its stake.
  - **FUNDED:** An opponent answered the proposal agreeing and locking the stake.
  - **SUSPENDED:** A challenge before is undergoing and must be answered.
  - **CLOSED:** The game concluded and the stake was given to the winner.
  - **CHALLENGED:** The game concluded due to a faulty exit being challenged.
- **ForceMoveChallenge** representing a current force move request if any.
- **Exits** ExitData for all tokens used in the channel.
- **Challenges** representing all challenges against the channel's exits.

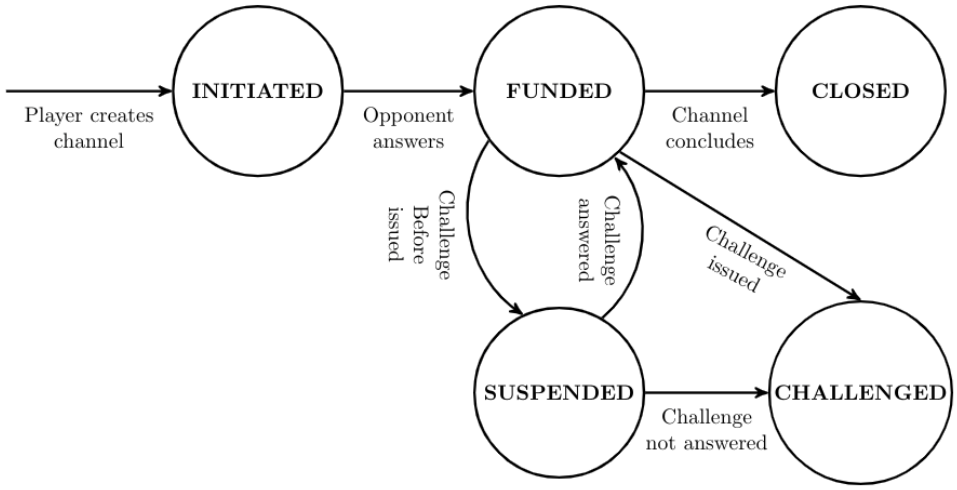

Figure 5.1: Channel States

5.3.3 Channel Fundation

In order for a channel to be funded, both parties must agree on the initial state and deposit the stake. For this one user *(from now on known as player)* will call the function on the **Channel Manager** which initiates a channel. This function needs to validate the *channelType* to be correct, the *stake* to be deposited to the **Channel Manager** and the *initialGameAttributes* to be a valid start state with the **isValidStartState** function on *channelType*. Providing to this function the **ExitData**, the corresponding Exit structures are generated for the player's tokens.

```
function initiateChannel(
      address channelType,
      address opponent,
      address key,
      uint stake,
      bytes calldata initialGameAttributes,
      bytes calldata exitData
   ) external payable;
```

```
function validateStartState(
      bytes calldata state,
      address[2] calldata players,
      uint exitDataIndex,
      bytes calldata exitData
) external view returns (RootChain.Exit[] memory);
```

The *exitDataIndex* indicates whether the **ExitData** corresponds to **the player** or **the opponent**, as each one provides their own **ExitData**. At this stage, the channel is in the **INITIATED** state.

For this channel to be used, **the opponent** must then fund the channel, accepting the conditions and providing its share of the stake. It also provides the key he will be using, as well as the **ExitData** necessary to create the exit structures.

```
function fundChannel(
      uint channelId,
      address key,
      bytes calldata initialGameAttributes,
      bytes calldata exitData
) external payable channelExists(channelId);
```

The *exitData* required for **the opponent**'s tokens are provided as well as the public key to check signatures. Once this function ends correctly, the channel is now **FUNDED** which means the stake is secured and the players can proceed to play starting from the initial state.

5.3.4 Channel Turn Transition

The main advantage of using a game channel is the ability to have computation off-chain, which is free. In a PVP game, the outcome of the game is calculated on both users' machines. However, this comes with a myriad of inconveniences as there is no way to trust the party on the other end of the channel. For this, users will take turns deciding their move, signing the states to the other party, so that both

users have proof on an agreement.

States contain the following structure:

- **channelId** the Id of the channel being used.
- **channelType** the address of the game rules.
- **participants** the players signing the states.
- **turnNum** the number of the turn.
- **gameAttributes** the serialized game state

When a state is created, the hash of the state is calculated using all of the previous values. This hash is then signed with the **private key** generated by the user for this game. The *channelId* and *turnNum* values served as a prevention of a repetition attack. A state signed by a player can't be used later on or on another game, as the *channelId* and *turnNum* won't match.

After a player transitions the state and signs it, this new state is now sent to the other player. The second player receives it, and now is his turn to transition to the next state, signing it. This goes back and forth between the parties until an end state is reached. Every odd-numbered state will be signed by one player and every even-numbered state will be signed by the other. Once a player receives a signed state, the result of any valid transition that player decides to do is considered a valid and agreed new state. This means that any consecutive pairs of states, given the condition that their transition is valid, and their signatures, can be considered as a consensus between the two parties of the game state at some point.

The transition of the state is always the same: ChannelId, channelType and participants remain the same and turNum is increased by one. However the transition from one gameAttributes to another is more complex and is validated on a game-to-game basis. As an example we'll consider a turn-based game where each player makes a decision concealed from the other, until both are revealed at the same time and they result in some outcome.

Lets define some *gameAttributes* as an *RLP-encoded* array of values, where the first values represent **The Game State**, the values that are changed after each round.

The **Initial Game State** only contains the initial values that were agreed in during the creation and funding of the channel. This will be the starting point for the game. Since the second player, **the opponent**, is the one funding the challenge, he will be the first to make a transition.

Each player has to make a decision, however this decision must be concealed so no player has a decision advantage, as both decisions will trigger simultaneous, just like in *Rock-Paper-Scissor*. This comes with an issue due to the fact that players take turns signing the states, and there is no way to have them simultaneously sign something. For this, a commit-reveal mechanism is used. If **the opponent** makes a decision and hashes it, it can easily provide the hash to **the player**. **The player** can now make a decision without knowing **the opponent**'s decision, but it can know for sure that **the opponent** already made a decision and it cannot be changed, since changing it won't validate with the provided hash. After that, **the opponent** can simply reveal his decision, also proceeding with any calculations necessary now that both decisions are known.

Since the decision pool is limited, just hashing it wouldn't be secure enough, since an easy way to know what decision was made is just hashing all possible decisions and check by equality which decision resulted in the provided hash. To avoid this, a random salt can be generated when the decision is made, and then hash these together. This way, there is no way to brute-force the correct decision, and at the end, the salt is provided with the original decision for hash verification.

This salt can be used for other purposes as well, such as random number generation. Since this is a PVP environment, there is no way to get a trust full random number, since one of the parties will calculate it, and nothing stops it from cherry picking a favoured number. The workaround of this is getting some entropy from both parties, if both parties provide part of the entropy, then the random number

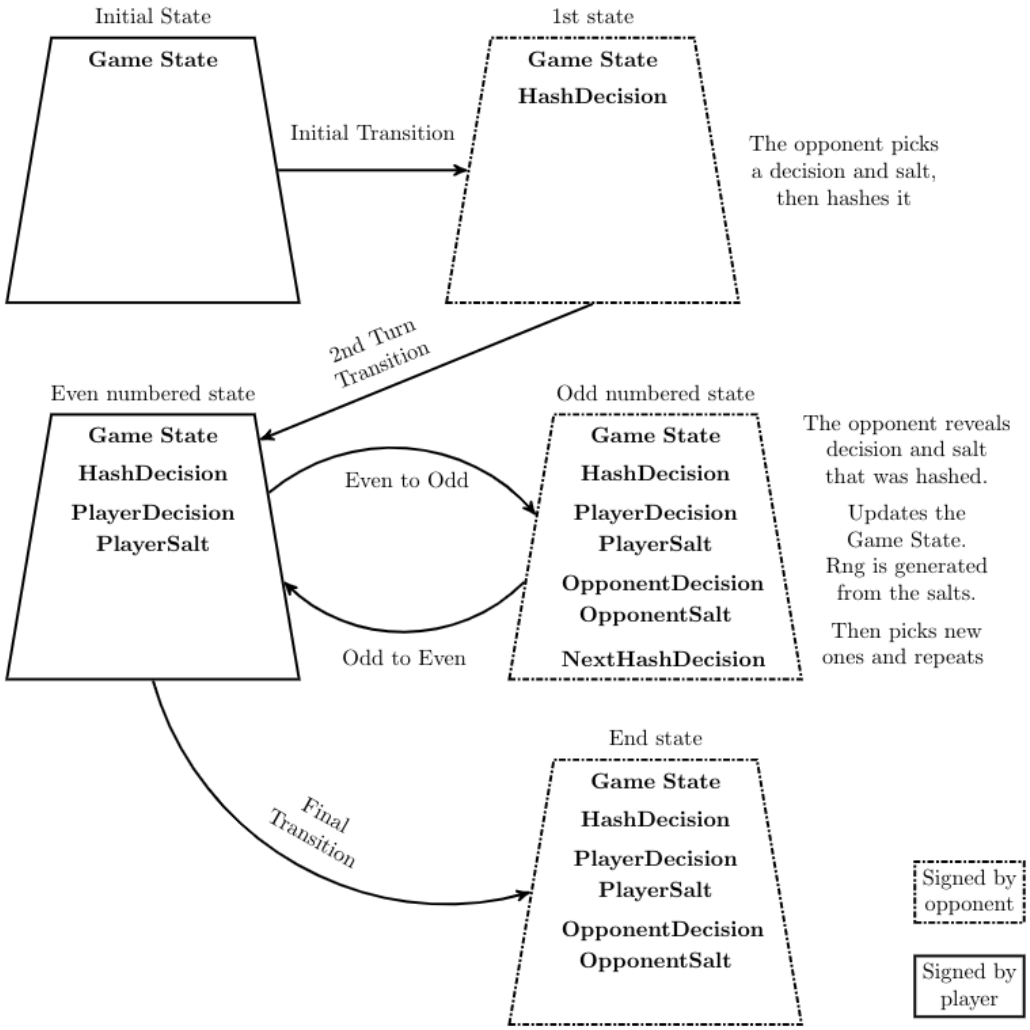

Figure 5.2: Game Transitions

will be known and it can't be selected by one of the two. However, it fails under the same issue with the simultaneous decisions. If one party selects part of the entropy, the other can iterate during many possible entropy counterparts until one generates the desired number, selecting them. But when using the salt to obfuscate the decision, then **the player** can select a random salt knowing that **the opponent**'s salt was already selected and can't be changed, since that would invalidate the hash making the transition invalid.

This back and forth of rounds, with simultaneous decisions and randomness, can go indefinitely until an end state is reached. Once a state can be considered finalized,

the winning user can provide the necessary data to the **Channel Manager** to unlock and retrieve the bets, all of this without having to pay for any intermediate computational fee during the game lifespan.

5.3.5 Channel Conclusion

In order to close a channel (fig. 5.3), the **Channel Manager** needs proof that both parties agreed on a conclusion. For this, the following function is provided:

```
function conclude(
      uint channelId,
      State.StateStruct memory prevState,
      State.StateStruct memory lastState,
      bytes[] memory signatures
) public channelExists(channelId) isFunded(channelId);
```

By providing the two last states of the channel, and their corresponding signatures, the **Channel Manager** can, validating the correct transition between the two, know that an agreement was arrived. There can't be a way for two consecutive states, signed and with a valid transition, to not be an agreed game state. If the **winner** is the player who signed the *prevState*, then *lastState* was signed by the other party, so it agrees with it. If the **winner** however signed the *lastState*, then the other party signed the *prevState* and, since the transition is validated on the blockchain, therefore agrees to the *lastState*.

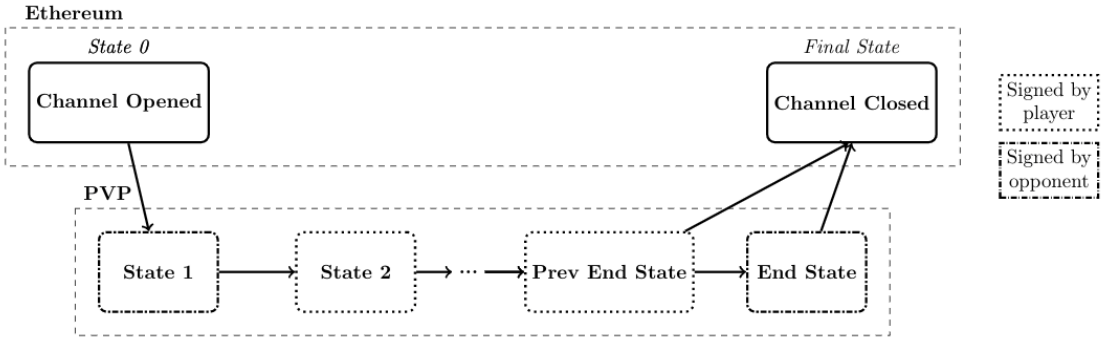

Figure 5.3: Channel Conclusion

5.3.6 Force Move Challenge

With most off-chain channels, such as the **lighting network** in Bitcoin and **raiden network** in **Ethereum**, users can opt-out at any moment from the channel, splitting the funds according to the current state. Taking a closer look, that does not apply to this case, since a game channel there is no way of knowing **the winner** of a channel unless that channel reached the final state.

The issue comes when a player sees their irreversible loss ahead of them and refuses to answer a state, or simply disappears, there needs to be a way to force some player to make a move, or else, to forfeit the channel. The idea of **Force Move Channels** was conceptualized [10] so that the only trusted intermediate is the decentralized blockchain, and it must be the one forcing a player to make a move. At any point a player can, providing the last two known states, force a move calling the following function.

```
function forceMove(
      uint channelId,
      State.StateStruct memory fromState,
      State.StateStruct memory toState,
      bytes[] memory signatures
) public channelExists(channelId) isSenderAllowed(channelId);
```

Given this **game state** as a starting point, the challenged player must provide an answer to the **Channel Manager** in a 10-minute window, or else the channel is forfeited and the challenger receives the rewards. If the challenge is answered, then the game can continue from that point onward. This completely removes the issue of a non-responding party.

```
function respondWithMove(
      uint channelId,
      State.StateStruct memory nextState,
      bytes memory signature
) public channelExists(channelId);
```

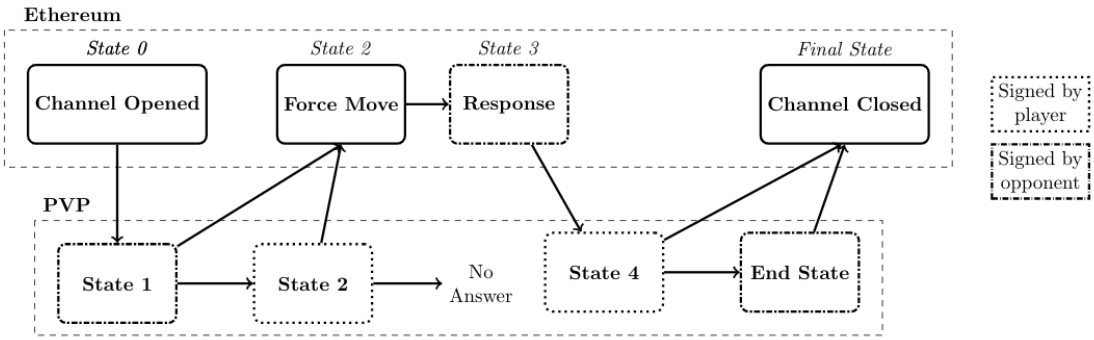

Figure 5.4: Force Move Challenge

This raises one more issue of having a challenge be issued to an old or a branching state. If one of the users decides some path the game is taking is not of his convenience, using an older state signed by his opponent, he could sign a response to that, altering the past, and forcing a move on those states. However, if the opponent answers, then it will agree to that **game state**. To counter this the following function is provided.

```
function refuteChallenge(
      uint channelId,
      State.StateStruct memory refutingState,
      bytes memory signature
) public channelExists(channelId);
```

To **refute** a challenge, a *refutingState* can be provided, which is any state signed by the challenger that has a higher **turnNum** or equal **turnNum** and different hash. If there is such a state, that means the challenger issued a challenge with an old or a different state and can be punished for that, loosing his bet and closing the channel.

5.3.7 Plasma Chain Challenge

For every channel, a user must declare which tokens will be used for the **game**. Since there is no way for the **Channel Manager** to know who the true owner of a token is, as they are deposited in **Plasma**, the same procedure as when exiting a

slot can be used. When creating the channel, the exit data is provided and an **Exit Data Structure** is generated for those tokens. If any player attempts to start a channel using another user's token, just as trying to exit another user's slot, the exit data provided is challengeable. These simulated exits can be challenged by anyone as long as the channel is still open, and if succeeded, then the bets of the fraudulent participant will be provided to the challenger, and the rest released to the owner. The functions to challenge look a lot like the ones to challenge an exit, with one key difference. Since there are many **Exit Data Structures** per channel, an index is indicated to which of them is the one being challenged.

```
function challengeAfter(
      uint channelId,
      uint index,
      bytes calldata txBytes,
      bytes calldata proof,
      bytes calldata signature,
      uint256 blockNumber)
external channelExists(channelId) isFunded(channelId);
```

```
function challengeBetween(
      uint channelId,
      uint index,
      bytes calldata txBytes,
      bytes calldata proof,
      bytes calldata signature,
      uint256 blockNumber)
external channelExists(channelId) isFunded(channelId);
```

One main difference is the fact that no transaction created after the foundation of the channel can be used as a challenging transaction. The requirement is to be the owner of the token at the start of the channel. This streamlines the challenging cases and removes some complications, while keeping the essence of **Plasma**.

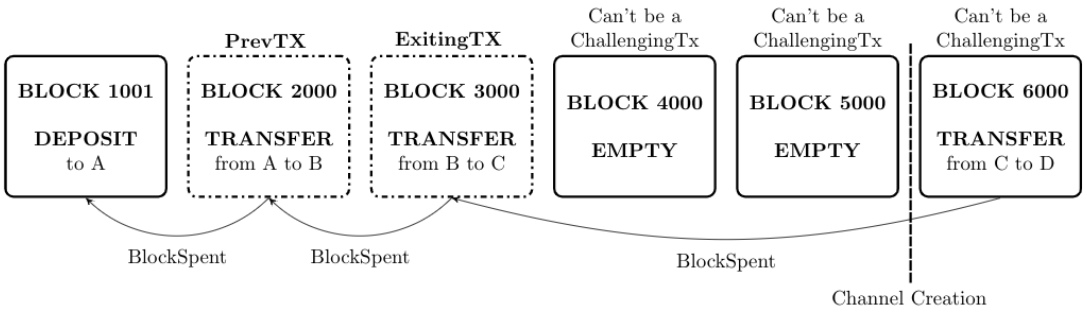

Figure 5.5: Channel Challenge After

Just as when challenging an **exit**, a *Challenge Before* can be issued. If this is the case, the same mechanism is used, where the challenger must provide a **bond** that if responded, will be lost. A channel may contain many unanswered challenges, just as before, they are removed from the queue as they are answered. When a channel contains a **Plasma challenge**, its state is changed to **SUSPENDED**. While in suspended, the channel cannot be interacted with. No force moves, and no conclusions until all the challenges are answered. If after a day the channel's challenges wasn't answered, the fraudulent user's stakes are given to the challenger.

```
function challengeBefore(
      uint channelId,
      uint index,
      bytes calldata txBytes,
      bytes calldata proof,
      uint256 blockNumber
) external payable channelExists(channelId) isChallengeable(channelId)
   Bonded;
```

```
function respondChallengeBefore(
      uint channelId,
      uint index,
      bytes32 challengingTxHash,
      uint256 respondingBlockNumber,
      bytes calldata respondingTransaction,
      bytes calldata proof,
      bytes calldata signature
) external channelExists(channelId) isSuspended(channelId);
```

Conclusion

Plasma and other scalability solutions are still in the early stages of development. In the cryptocurrencies world, decentralized applications are a small share of it, and decentralized gaming almost a niche. However, in the past couple of years, more and more interest has been put into decentralizing gaming since it adds some clear value to the experience.

A developer must be sure of the technology being used and the purpose of it before venturing into creating something on it. While the term **blockchain** is greatly used nowadays, it is usually misused as a way to attract interest solely because of its revolutionary aspect. The fact of the matter is, **blockchain** contains more flaws than strengths. Is costly, ineffective, slow and troublesome to work on.

Nevertheless, where **blockchain** succeeds in, it does it in spades. No other technology has been able to provide the level of ownership and accountability that **blockchain** accomplishes. For this, only applications that exploit this advantage are truly worth developing on **blockchain**.

Games are usually not the case. Historically, games were regarded as media software, whose sole purpose was entertainment. A single-player experience with no real-life impact has no way to make use of **blockchain**'s strengths. However, gaming applications have been evolving at an increasing pace, reaching the highest grossing media industry, and taking on shapes that were unthinkable when the first **Pong** game was conceptualized.

Going into a world of **game as a service**, an always connected gaming mentality, having games provide assets and rewards with scarcity, such as the **Steam**

Marketplace where items can be bought and sold, with people living of not only developing but also playing games, there is a line not too far in the future where **blockchain** and **gaming** can meet. For this, pushing the technology forward is a must, as having the technology available is the first step into opening design spaces for developers to create on.

Plasma is not the all-around solution we had in mind when starting this project. When digging into it, its limitations are clear. Only transfers are possible on it, users have to be constantly checking for a possible steal and the process and integration into the market is difficult. But even the many flaws **Plasma cash** has, it is considered a great step in the right direction.

With this project, a real-life game where players can truly feel owners of their assets, can be achieved with minimal costs. Being able to have player's trust on the actual code being run, improves some aspects of the game such as involving real-money that players are mindful of.

The technology has still a long path to go before it is used at a large scale, and many improvements and scalability solutions may be found in the future. However, the findings that were shown in this project serve as a good base from where to work on and a good glimpse of what can be achieved with it.

Appendix A - Implementation

In order to showcase an example of all the described technology, **CryptoMonBattles** was developed. The project is divided in three main components: the **Root Chain**, all the deployables contracts to the blockchain. the **Side Chain**, a backend in charge of block creation, run by **the operator**, and the **Client**, a web interface with which users will be able to interact. These components work together in order to provide the player a truly decentralized gaming experience.

7.1 Game Rules

There is a total of 151 different **CryptoMons** species with their own **base stats** and types, however every instance of them has its own **additional stats** that make them *truly unique*, there is even 1/1024 chances to have a special colored **CryptoMon**.

While the species of the **CryptoMon** define its looks, type and base stats, each **CryptoMon** is unique in health points, attack power, special attack power, defense, special defense and speed. Each **CryptoMon** can be of one or two of the following **types** based on the specie: Normal, Fighting, Flying, Poison, Ground, Rock, Bug, Ghost, Steel, Fire, Water, Grass, Electric, Psychic, Ice, Dragon, Dark and Fairy.

In **CryptoMonBattles**, battles have a **turn-based** system where each player selects a move and then calculations are made to reflect these decisions on the **CryptoMon**'s health points. When one of the **CryptoMon**'s health points reach 0, the battle ends, awarding the winner to the surviving **CryptoMon**.

When deciding a move, a player must select from the following pool of choices:

- Attack (one for each type)
- Special Attack (one for each type)
- Status (one for each type)
- Protect
- Shield Break
- Cleanse
- Recharge

Each player has a maximum of 3 **charges** that will be consumed when using any move other than *Recharge* or *Protect* and a charge can be acquired by using the move *Recharge*. For this players must be cautious on when the best time to recharge is, since your opponent will have the advantage on that turn.

Both *Attack* and *Special Attack* inflict damage, although they use a different formula to calculate it involving different stats of both **CryptoMons**. There are also five levels of **effectiveness between types**, as some attack types are or less effective against other defensive types. For example a fire attack to a grass **CryptoMon** is much more effective than against a water one. On top of that, there is a chance for a **critical strike**, that will amplify the damage by 50%.

The *Status* move is named differently according the the **CryptoMon's** type and its effect varies. For example, the grass status move is called *Leech Seed*, which absorbs health points from the opponent and restores life by the same amount. *Hurricane*, the flying status, hits the opponent at the end of each turn but also increase the player's **CryptoMon** speed by 50%.

It is possible to use *Protect* to nullify an opponent's attack, making him waste a charge, but if the opponent uses the *Shield Break* move in the same turn then the **Cryptomon** takes damage equal to 30% of its maximum health. This attack prevents people from spamming *Protect* and making the battle non-interactive.

7.2 Root chain

Several contracts were created and deployed to an **Ethereum** blockchain. The key ones are the *RootChain*, which servers as the **Founding Contract** of the **Plasma Chain**, *CryptoMon*, which is in charge of creation and distribution of the **CryptoMons**, *PlasmaCM*, also known as the **Channel Manager** and *CryptoMonBattles*, which contains the necessary logic for all **CryptoMonBattles**' calculations. The contracts are written in **Solidity**, a language that can run on top of the **Ethereum Virtual Machine**. The mentioned contracts, alongside with some other libraries, dependencies and interfaces, enclose all the functionality the project needs, described in previous sections. *RootChain* is in charge of all basic **Plasma** functionalities like deposits, transfers, exits, challenges and also the added swap feature. The other two contracts, *PlasmaCM* and *CryptoMonBattles*, manage the Force Move Channels for starting and concluding battles, besides handling the battle's challenges.

For testing purposes, a local blockchain was simulated and all contracts were deployed locally. This local blockchain can be run using **Ganache CLI**[1], and it can be configured to run in a **deterministic** way by setting a mnemonic, that means in every execution the tokens, accounts and transactions hashes will be the same. This feature combined with a debugger allowed the development to be more streamlined.

7.3 Side chain

Since **Plasma Cash** allows users to not trust the operator, the implementation of the side chain is a **centralized API run by a unique node**. The **API** is implemented in **Node.js** and persists all the needed data in a **Mongo** database. As defined by Plasma, **the operator** has complete control of what happens inside it. The side chain must also listen to events from the blockchain to update the stored information accordingly.

[1] `https://github.com/trufflesuite/ganache-cli`

Deposits

Every time a deposit is made to the CryptoMon contract an event is emitted and listened by the operator. A new block is immediately created with the deposit transaction. Deposits blocks' block numbers are always between two regular values of the regular increment, that means that `deposit_block_number \% 1000 != 0`, being 1000 the regular increment for any non-deposit block. A deposit transaction is easily identified because the previous owner of the token is the address `0x0`.

Transactions

Periodically, every 20 seconds by default but configurable, a block is mined with the waiting and valid transactions inside it and its *RootHash* is submitted to the **RootChain**. Each one of these blocks' number is a multiple of the regular increment, in this case 1000, meaning that `block_number \% 1000 = 0`.

Swaps

When the side chain receives a swap request and its signed confirmation, then it is able to include both transactions inside the same block. However, these transactions are invalid until both secrets are revealed. When all of the block's secrets are revealed, and **the operator** validates them successfully, then a *RootHash* to the **Secret Revealing Chain** is submitted and the previously submitted transactions representing the swap are valid.

Proofs and history

The operator provides anyone the neccesary proof and history for any **slot**. The proof will contain all the necessary information for a **Merkle Proof** and check the inclusion or non-inclusion of the slot inside the block. If anyone needs to verify the history of a **slot** the side chain provides the proof and transaction bytes for every block since the deposit, the requester should then check the inclusion and non-inclusion of the slot in every one of them to validate the history and confirm the ownership.

Battles

When two players battle, there is no need for a third party to be involved, since the calculations occur on both ends of the connections. But in favor of simplifying process, **the operator** acts as an intermediate for two reasons:

- To avoid implementing PvP battles (i.e. a peer to peer connection between players), each one of them connects to the **API** vía websocket which just acts as a middleman validating turns and sending to the opponent the new state of the battle.
- To allow disconnections and reconnections. The **API** knows every state submitted by the players and store the last two states (two states are needed for a **force move**). When one of the players reconnects to the battle, he immediately receives the last two states of the battle and he is able to continue with it, either by providing the next move, emitting a **force move** or wait for the other player to finish its turn.

Because the operator works as a middleman of the battle, he also listens to events emitted in a **force move** and **force move response** to update the battle state correctly since these moves are sent directly to Ethereum.

Other features

Because the side chain has all the information about transactions and it is listening to events from Ethereum it is able to know when an unlawful exit is being made, therefore it will automatically **challenge** any misbehaviour and probably be the first one to do it and receive the corresponding bond.

7.4 Client

The client is a web-based user interface implemented with **React** that allows players to interact in a human way. It is in charge of showing visual representations of the **CryptoMons** and interact with both the **SideChain** and the **RootChain** (for transactions, swaps, exits, challenges, start battles, etc.). The client also listens to events emitted by contracts like a new block being mined, swap and battle requests among other things.

Frequently, when there is a need to call certain contract methods or sign a payload, the user must use its private key. For easier management of private keys and signatures, the client relies on an extension that manages all those tasks like **Metamask** [2]. Each time the private key is required, a Metamask popup will appear with the corresponding action that requires the user's attention.

Plasma capabilities

The client application allows to deposit, transfer, swap, exit, challenge and validate the history of tokens. For the first three, there will be a direct interaction between the client and **the operator** or the contracts as detailed in previous sections. since the client doesn't keep track of every submitted block, to validate the history of a **slot** it must request to **the operator** the necessary proof for it.

When doing exits and challenges, in order to generate the **exitData** required, the client fetches this information from **the operator**. In an ideal scenario, this kind of information should be kept offline, ready for access in case of need. However, due to scope constraints, the project relies on **the operator** to provide it.

Battles

As mentioned before, battles must be started and concluded with calls to the blockchain. The interaction can be done peer-to-peer without **the operator**'s intervention. However, in this implementation the operator acts as an intermediator between the clients connecting to them via **WebSocket**. The clients provide state updates where every new state must be signed before submission. To avoid signing via Metamask for every move, a new public/private key pair is generated, which is then used to sign every state automatically, to provide a seamless interaction with the interface.

[2] `https://metamask.io`

Hacks

The client contains a section called "Hacks". In this section it is possible to act as a **malicious user** and introduce **operator mistakes** (or misbehaviours). The available hacks are:

- **Force old exit:** If the user used to have in its possession a **slot** but then transferred it, this **slot** is eligible to an old exit. An exit attempt is done providing the old and expired **exitData** when the user was the owner. This hack is easly challengeable with a **Challenge After**.
- **Create double spend exit**: If the user used to have in its possession a **slot** but then transferred it, a new transaction to itself can be created, spending the block that was already transferred. This generates a double spent on that block, and an exit can be generated with this **exitData**. This hack is challengeable with a **Challenge Between**.
- **Create non-existent transaction and exit**: As the name implies, this hack will create two transactions transferring the token from the user to itself. The first transaction, since is not validated in the exit, comes from an unknown source. This hack is challengeable with a **Challenge Before** without any chance to respond with a valid counter proof.

There are similar hacks to start battles with a not-owned token and they can be challenged similarly. Finally, in this section it is also possible to start a challenge before that will be respondable.

7.5 Future improvements

Due to scope limitations, some features had to be put aside and not implemented, which help reduce the trust needed on **the operator** while playing this game.

The first feature that would decrease the trust put into the operator is an **offline client** that should listen to events from the blockchain and keep track of all mined transaction. With that information, this offline client would be able to create its own proofs to validate histories or to make an exit without the need of the **Side**

Chain. Currently, if **the operator** decides to hide information to prevent a user from exiting one of his tokens, it can do so.

Moreover, the **SideChain** stands between both parties during battles and stores the information of the last two state of each battle. However, this means that the client only works correctly if the operator is behaving correctly as well. Clients should be able to establish a **peer-to-peer connection** without the need of a third party, only submitting to the **RootChain** when needed.

Another improvement to the project would be to provide the ability to **punish the operator** on misbehaviours. With the current implementation, the only incentive for **the operator** to act righteously, is to avoid a massive exit due to loosing its users trust, rendering the **SideChain** pointless. One of the proposals is to freeze a large bond into the **Founding Contract**, and in case of misbehaviour, **the operator** will loose it automatically.

Appendix B

8.1 Merkle Tree and Sparse Merkle Tree

A Merkle tree is a data structure conceptualized by *Ralph Merkle* in 1988 [11]. He presented a data structure with the ability to provide proof of existence and non-forgery of an unlimited amount of messages, while reducing it to a single hash output. For this to work, the tree is defined as follows:

$$MerkleTree = \begin{cases} Hash(Message_i) & : \text{for any } Leaf_i \\ Hash(LeftChild||RightChild) & : \text{for any Parent Node} \end{cases}$$

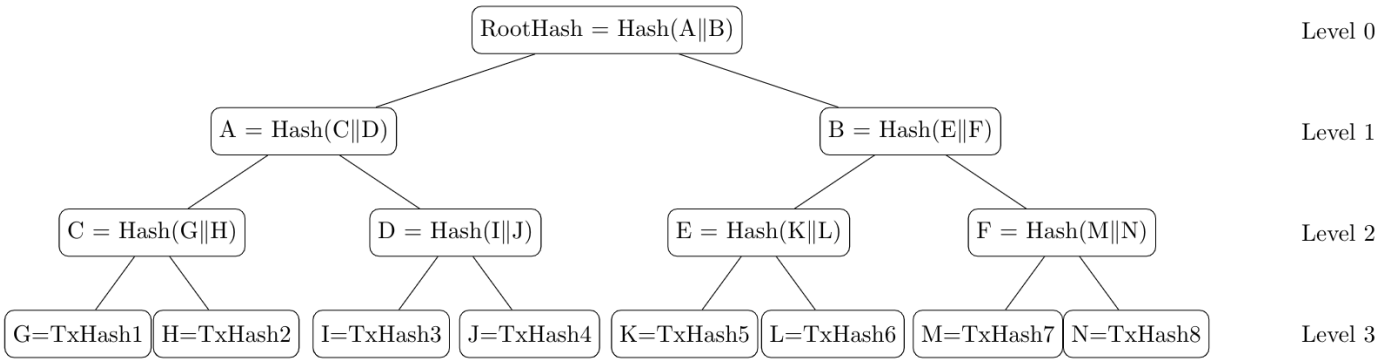

Figure 8.1: Merkle Tree

The root of this tree, also known as the *RootHash*, is then provided for future verification. This is extremely useful in blockchain as it means there is an unlimited amount of messages that can be included in a block by providing a single hash

output. These messages usually translate as transactions.

For this to work, the interested user would have to gather all the leaf messages, and their order, and re-create the *RootHash*. If the *RootHash* generated is the same as the one provided, then there was no forgery done at any of the transactions. However, it is not efficient for a user to recreate the whole tree for a verification of a single message. For this, a **Merkle proof** can be provided, which if successful, proves the inclusion of a message in a Merkle Tree. To validate the proof the user receives the inner nodes necessary to recreate the path to the root, starting from the interested message. This proof involves every sibling of the nodes in the path to the root. These siblings are then concatenated with the intermediate calculated hashes to generate the parent node, until the root is reached and the proof can be validated.

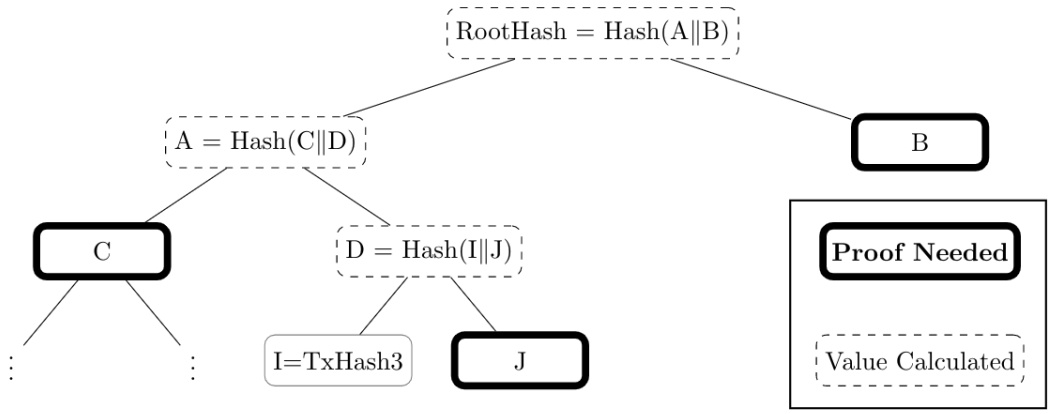

Figure 8.2: Merkle Tree inclusion proof

In Bitcoin and other blockchains, Merkle Trees are used as a way to store validation of transaction inclusions to a block. If a proof can be verified against the *RootHash* of the block, then the transaction can be considered as included in that block. However, if a proof can't be validated there can't be a way to tell whether the transaction is or is not included. With standard merkle trees, only inclusion is provable, while omission is not. The Merkle Tree to be used during this project

must have the ability to validate omission of transactions. For this, a **Complete Merkle Tree** is required.

$$Complete\ M.Tree = \begin{cases} Hash(null) & : \text{for any } Leaf_i \text{ where } Message_i \text{ is not included} \\ Hash(Message_i) & : \text{for any } Leaf_i \text{ where } Message_i \text{ is included} \\ Hash(LeftChild||RightChild) & : \text{for any Parent Node} \end{cases}$$

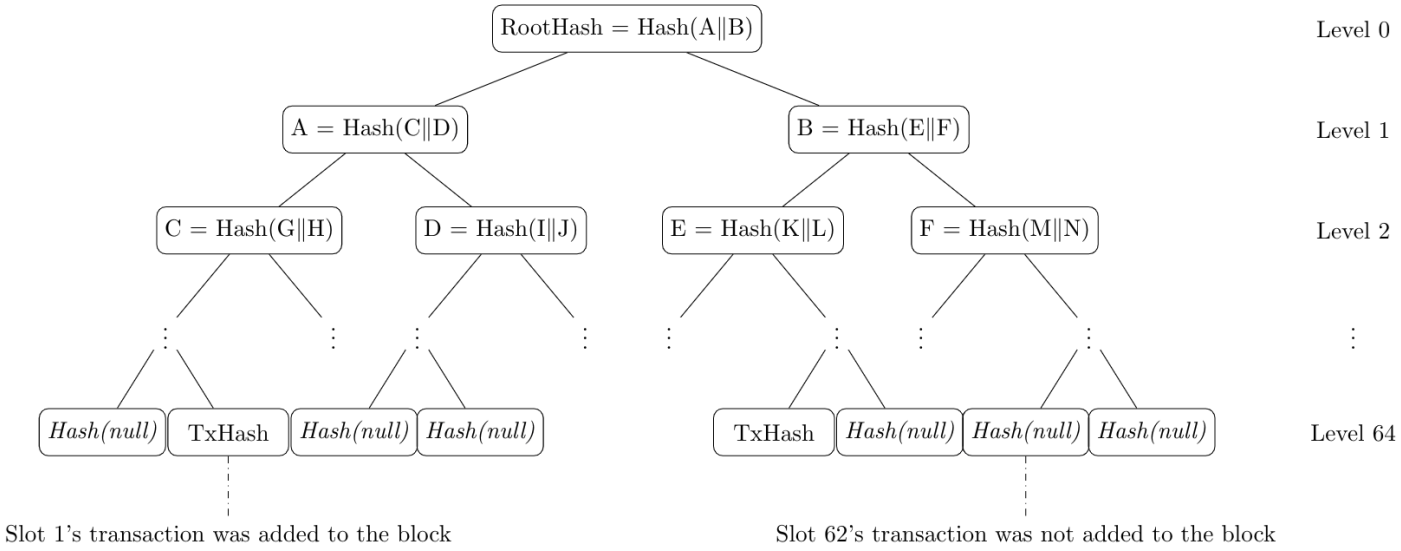

Figure 8.3: Complete Merkle Tree

A complete Merkle tree of n levels is a tree able to contain up to 2^n messages where each leaf is a slot available for a message. Then, a convention is needed for a message to self-define in which slot number it must be present for it to be valid. This way, a **Merkle Proof** could be used to prove its inclusion against the message itself, or prove its omission against *Hash(null)* in that slot.

On the downside, a Complete Merkle Tree is a huge data structure. A Complete Merkle Tree of 2^{64} slots requires an outstanding $2^{65} - 1$ hash computations, an amount that, if possible, would nullify the hash function all together. For that, an intrinsic value of a Complete Merkle Tree can be used as an advantage: Repetition. The fact that any empty slot uses the same value (*Hash(null)*) makes it obvious how

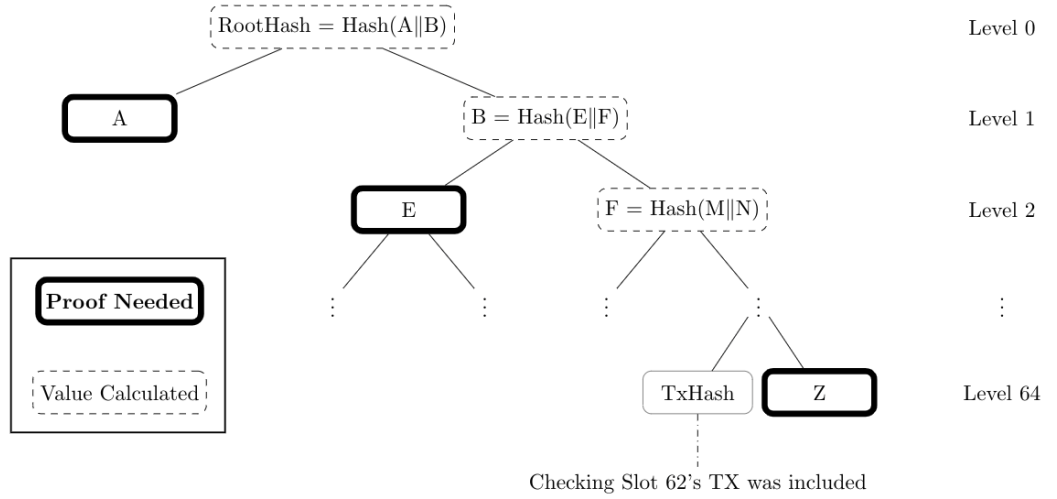

Figure 8.4: Complete Merkle Tree inclusion proof

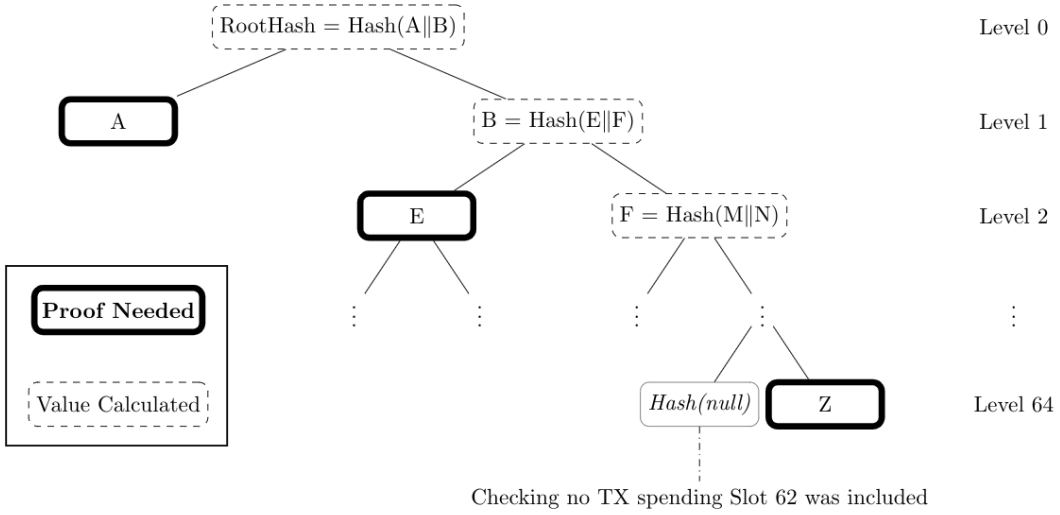

Figure 8.5: Complete Merkle Tree omission proof

complete branches can be pre-calculated if all of their leaves are empty. Defining the following Sparse Hash function:

$$Sp.Hash^{n}(x) = \begin{cases} \text{Hash(x)} & : n = 0 \\ \text{Hash}(Sp.Hash^{n-1}(x)||Sp.Hash^{-1}(x)) & : n > 0 \end{cases}$$

With this function, the *Sp.Hash* of each level can be easily calculated. When creating the *RootHash*, only the messages added to the tree are required. Whenever a node is

needed, whose branch was not taken into account, the *Sp.Hash* of that level can be calculated to get its value. This also reduces the proof size, as many of the required nodes to recreate the path are also instances of *Sp.Hash*, which can be indicated with a single boolean.

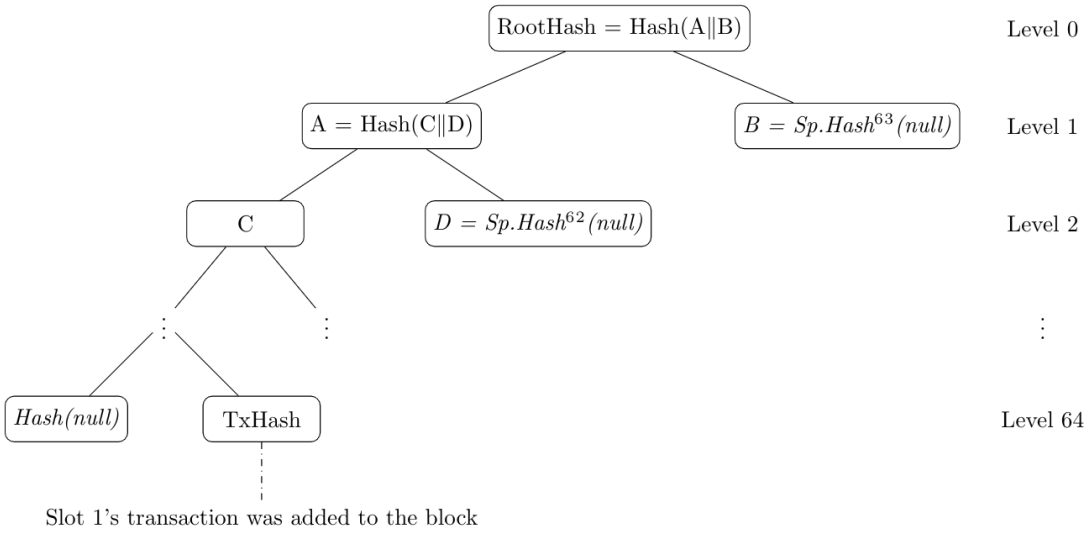

Figure 8.6: Sparse Merkle Tree

Bibliography

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system", http://bitcoin.org/bitcoin.pdf, 2008.

[2] V. Buterin and D. G. Wood, "Ethereum white paper: A next-generation smart contract and decentralized application platform", https://github.com/ethereum/wiki/wiki/White-Paper, 2013.

[3] J. Poon and V. Buterin, "Plasma: Scalable autonomous smart contracts", https://www.plasma.io/plasma.pdf, 2017.

[4] V. Buterin, "Minimal viable plasma", https://ethresear.ch/t/minimal-viable-plasma/426, 2018.

[5] V. Buterin, "Plasma cash: Plasma with much less per-user data checking", https://ethresear.ch/t/plasma-cash-plasma-with-much-less-per-user-data-checking/1298, 2018.

[6] D. Robinson, "Plasma debit: Arbitrary-denomination payments in plasma cash", https://ethresear.ch/t/plasma-debit-arbitrary-denomination-payments-in-plasma-cash/2198, 2018.

[7] W. Entriken, D. Shirley, J. Evans, and N. Sachs, "Eip 721: Erc-721 non-fungible token standard", https://eips.ethereum.org/EIPS/eip-721, 2018.

[8] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "The keccak sha-3 submission", https://keccak.team/files/Keccak-submission-3.pdf, 2011.

[9] V. Buterin, "Plasma cash minimal atomic swap", https://ethresear.ch/t/plasma-cash-minimal-atomic-swap/3409, 2018.

[10] T. Close and A. Stewart, "Forcemove: An n-party state channel protocol", https://magmo.com/force-move-games.pdf, 2018.

[11] R. C. Merkle, "A digital signature based on a conventional encryption function", Advances in Cryptology — CRYPTO '87. doi:10.1007/3-540-48184-2_32. 1988.