Plasma Cash: Towards more efficient Plasma constructions

Georgios Konstantopoulos
me@gakonst.com

January 12, 2019
WORKING DRAFT

Abstract

Plasma is a framework for scalable off-chain computation. We describe and evaluate Plasma Cash, an improved Plasma construction which leverages non-fungible tokens and Sparse Merkle Trees to reduce the data storage and bandwith requirements for users. We analyze the cryptoeconomic exit and challenge mechanisms used to keep user funds secured, even when the Plasma Cash chain's consensus algorithm is compromised. A reference implementation is provided for evaluation. Finally, we briefly discuss further improvements that can be made to the Plasma Cash protocol such as arbitrary denomination payments, less user data checking, fast and optimistic exits.

Contents

1 Introduction

Plasma is a scalability solution to increase the throughput of a smart contract enabled blockchain by allowing interoperable transfer of assets between blockchains. Assets are deposited from the sending blockchain (*mainchain*), to the receiving blockchain (*plasmachain*). Contrary to two-way pegged sidechains [1], funds stored in *plasmachains* can safely be withdrawn to the *mainchain* through a dispute process where the mainchain is responsible for the safety of the coins, often referred to as an *exit*. Safety is maintained even if the plasmachain's consensus mechanism is compromised, as long as the mainchain's consensus mechanism remains secure. It should be noted that Plasma improves throughput[1] rather than latency[2].

Plasma Cash leverages non-fungible tokens[3] by making the funds deposited in the plasmachain independent and unique [2]. This results in a simpler settlement protocol for withdrawing funds back to the mainchain, by requiring less data checking on their coins, compared to the technique mentioned in the original paper [3]. Discussions around this technique have been found across the web, in ad-hoc discussions and videoconferences [4]. To date there exists no complete resource on the security and scalability guarantees of Plasma Cash.

1.1 Contributions

Protocol Analysis We present a full analysis of the Plasma Cash technique, decomposing it to its components, explaining parts of the protocol and its functionalities in detail. We additionally explain the various extensions which can be added, along with the benefits and tradeoffs they give.

Analysis of Attacks We discuss the currently known attacks on Plasma and how a Plasma contract should be designed to mitigate these attacks. With proper mechanism design and the attachment of 'cryptoeconomic' bonds on actions that may result in faults we realize a system that is secure against adversaries, even if the adversaries have full control of the plasmachain's consensus mechanism.

[1] The number of transactions that can be finalized every N seconds
[2] The amount of time before a transaction is confirmed
[3] A non-fungible token (NFT) is a unique token. Only one copy can exist for a specific NFT.

Implementation A reference implementation of Plasma Cash is provided at `github.com/loomnetwork/plasma-cash`. It provides support for Ether and ERC20 tokens [4], as well as ERC721 Non-Fungible tokens [5]. The Plasma contract is token agnostic and extending it to support more types of token standards is a trivial process.

1.2 Related Work

In this section we will briefly describe work that is being done towards scalability via other techniques, each with its own tradeoffs. There are two approaches to scalability. Protocol level scalability[6], often referred to as Layer 1, and scaling through computation via Layer 2. Plasma is considered a Layer 2 scalability solution.

Sidechains A sidechain is a blockchain which has its own independently secured consensus algorithm, and is pegged to another blockchain [1]. Value can be transferred from one blockchain to another by relaying SPV proofs[7]. Verifying that blockchain A's event has occured on blockchain B requires SPV proofs which grow linearly with the number of blocks[8]. It should be noted that users can only withdraw their funds back to the mainchain if the sidechain's consensus is secure and does not censor transactions or produce invalid state changes, meaning that the consensus mechanism is the sidechain's 'custodian'. Using a sidechain for scalability is thus limited by the security and decentralization tradeoffs introduced by increasing capacity and throughput.

Payment Channel Networks A payment channel is a mechanism which enables two parties to trade value in a rapid and feeless manner by exchanging signed messages which represent the latest state of the channel. The state can be settled on the underlying blockchain at any point in time. If a party attempts to settle an earlier state, the other party can challenge/dispute within a bound time period and penalize the fraudulent party. An on-chain transaction is required to initialize the channel. Channels can be unidirectional, bidirectional and can be linked together to create Payment Channel Networks, which allow two parties to trade with each other, even if they do not have a channel with each other, by routing payments through intermediaries. Payment channels feature instant finality since a payment can be considered complete the moment signed attestations about it are exchanged from the participating parties. In addition to the on-chain funding transaction required during the initialization of a channel, Payment Channel Networks (cf. Lightning Network, Raiden Network) require a capital lockup equal to the value being transferred, as well as potential fees paid to intermediaries for routing payments.

[4] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

[5] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md

[6] E.g. sharding and bigger blocks

[7] Simple Payment Verification: Proofs that allow clients to verify the inclusion of a transaction in a block by verifying the block headers instead of downloading the whole block

[8] Recent research has shown that the proof growth can be reduced to be logarithmic [5]

Generalized State Channels Extending the idea of payment channels, generalized state channels tackle the problem of performing arbitrary computation off-chain, with the ability to settle on-chain, e.g play a battleship game off-chain, with the ability to settle a dispute about the general state of the game on-chain [6, 7].

2 Plasma Fundamentals

'Classic' Plasma The initial vision of Plasma describes a mechanism which enables connecting blockchains to a base blockchain often referred to as the 'rootchain' or the 'mainchain' [3]. The mainchain acts as the final arbitrator in the case of disputes as shown in Figure 1. Trees of plasmachains forming a hierarchy of blockchains similar to the court system would also be possible such that a dispute can be escalated all the way to the supreme court (the mainchain)

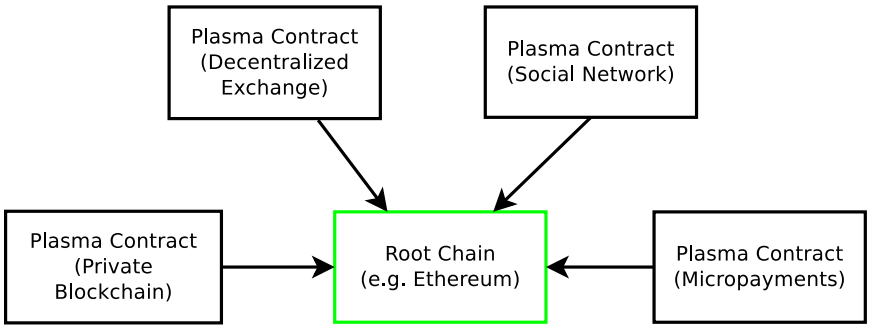


Figure 1: Multiple blockchains connected to a 'mainchain' [3].

A plasmachain that utilizes a potentially centralized consensus algorithm (e.g. Proof of Authority) derives its security from the mainchain's consensus algorithm. This is achieved by publishing a merklized[9] state root of each *plasma block* to the mainchain. The transaction model used is UTXO based, similar to Bitcoin[10].

Exits and challenges An *exit* is the mechanism by which a user expresses their intent to withdraw a UTXO from the plasmachain back to their account on the mainchain. After submitting the exit of a UTXO, the user needs to wait for a *challenge period* to pass, during which other users can *challenge* their exit, by providing proof that it is invalid[11].

A challenge is an attestation provided by another user, proving that the UTXO that is being exited is invalid or has been spent. Challenges can either be non-interactive, where the exit is instantly cancelled, or interactive where they require a response to the challenge

[9]The transaction set of each block gets inserted in a Merkle Tree, and the merkle root of that tree gets published to a smart contract.

[10]https://en.bitcoin.it/wiki/Transaction

[11]This is a common technique also used in payment channels, whereby invalid attestations can be challenged and the fraudulent user gets penalized if the challenge is successful.

before the *challenge period* is over. After the challenge period has passed, if there is no outstanding challenge, an exit can be finalized, giving its owner the right to withdraw the amount specified by the UTXO back to their mainchain account.

Exits and interactive challenges require users to attach a bond, as collateral. If a user's exit gets challenged and the challenge is valid, the collateral gets slashed and is sent to the user who reported the fraud. This incentivizes users to act honestly, and creates a crowdsourced system of watchers who are rewarded for reporting fraudulent activity.

3 Plasma Cash

Plasma Cash is a plasma construction with much less user data checking [2]. It utilizes Non-Fungible-Tokens (NFTs) to reduce the user checking requirements to only the NFTs that they own[12]. The system's security relies on users fully authenticating a coin's history before accepting it as a payment by utilizing Sparse Merkle Trees, which allow the efficient verification of inclusion and non-inclusion of a transaction in a block, as explained in the next subsection.

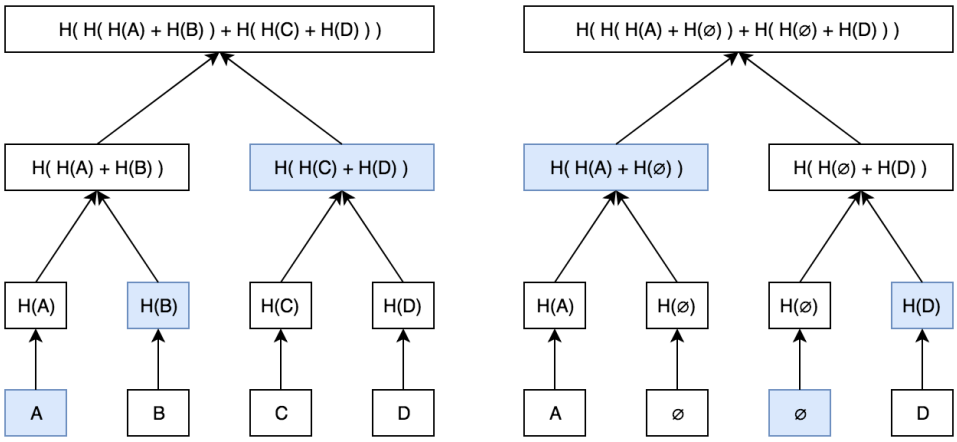
3.1 Sparse Merkle Trees

A Merkle Tree is a data structure which allows to succinctly commit to a dataset and prove the inclusion of a part of the committed dataset in $O(log_2(N))$ steps instead of $O(N)$, where N is the number of elements in the dataset, via *Merkle Proofs*. The committed value is called a *Merkle Root*. A Sparse Merkle Tree (SMTs) [8] is an ordered merkle tree, where each element of the dataset is placed at the leaf with the index corresponding the element's index in the dataset. If an element of the dataset was not included in the Merkle Root, its leaf is set to a special default value.

The inclusion of a transaction spending a coin in a block can be efficiently proven through a Merkle Proof. The same method can be used to prove the non-inclusion of a spend of a coin in a block. A coin can only be spent once per block because only 1 transaction at its slot can ever exist. If a coin was not spent, the leaf is set to the hash of 0. A visual representation of the above is given in Figure 2.

Further optimizations can be done to the verifier as suggested in [10] by precomputing the default values of the SMT and by introducing a bitfield in the proof which acts as a switch between choosing the next 32 bytes during the verification from the proof or from the SMT's default hashes. A reasonable estimation for a block containing 2378 transactions results in proof sizes being 320 bytes, compared to normal proofs which would be 2048 (64 * 32) bytes, for a SMT of size 64.

[12]Plasma-MVP requires users to be constantly monitoring the plasmachain for fraudulent state transitions, while Plasma Cash requires that users only watch the mainchain about fraudulent exits of coins that they own



(a) Merkle proof of inclusion for coin A

(b) Merkle proof of non-inclusion for coin C

Figure 2: Merkle Proofs in a Sparse Merkle Tree [9]

3.2 Periodic Merkle Commitments

All Plasma designs rely on a plasmachain operator that commits the Merkle Root of each generated block to the mainchain. If a Proof of Stake system is used, publishing a block must be accompanied by a number of validator signatures, exceeding a pre-agreed threshold. Whenever a Plasma Block's root is committed to the rootchain, all valid transactions in that given block can be considered finalized upon availability of the related witness data[13] for the inclusion of these transactions. Since only the block root is committed to the mainchain instead of all the included transactions, plasma can bundle together any number of transactions, the only limit to the number of included transactions being the size of the plasma block. The minimum finality time that can be achieved is the block time of the rootchain (15 seconds in Ethereum). Given that every block must be published to the mainchain, the operational costs for this process can become large. Operators can be expected to trade bigger finality time for less maintenance costs by committing block roots less often.

3.3 Non Fungible Tokens and Depositing a Coin

When value gets deposited in the plasma smart contract, a unique id and metadata key-value pair gets generated for that token and is saved in the contract's storage. The id is the unique serial number of the coin, and is what makes the deposited coin non-fungible. As a result, depositing 5 ETH two times creates two coins which have their own unique transaction history and are independent from each other. The unique id can also be thought of as a serial number, compared to fiat cash money.

[13] By witness data we refer to the merkle proof of inclusion of that coin in the specified block

After the coin gets saved in the smart contract, a block containing only the deposit transaction is appended to the plasmachain[14].

3.4 Transferring a Coin and Verifying its History

Each coin in Plasma Cash has its own unique coin history. A coin receiver must verify that the coin they are receiving has a valid history in order to accept it. A coin with invalid history is counterfeit and cannot be withdrawn safely. In order to validate a coin's history, a set of merkle proofs of inclusion and non-inclusion for the coin since its initial deposit must be sent to the receiver. The receiver can then proceed to verify that there were no invalid spends of the coin in the coin's history. This imposes a heavy storage and bandwith burden on senders that want to transfer a coin. Specifically, the proofs required to send a coin are $O(t * log_2(N))$ where t is the number of blocks since a coin's deposit and N is the number of coins the Plasma Cash chain supports.

In a real world scenario where a buyer wants to buy a product from a vendor the following is expected to happen, in a non-fraudulent case:

1. Buyer broadcasts transaction giving ownership of their coin to the seller
2. Transaction gets included in a block and witness data about its inclusion is made available
3. Buyer verifies that the transaction was included in the block
4. Buyer sends the proofs of inclusion and non-inclusion to the vendor.
5. Vendor verifies the history of the coin along with the correct inclusion of the coin's transaction in the block.
6. Vendor gives the product to buyer

A transaction is a tuple: `Tx(slot, parentBlock, newOwner, prevOwnerSignature)`. A transaction that was included in a block is a combination of the previous tuple and a merkle proof for the block: `IncludedTx(tx, blkNumber, proof)`.

The algorithm for verifying the history of a coin is given in Figure 4

3.5 Exiting and Withdrawing a Coin

As described in Section 2, exits are the mechanism by which a coin can be withdrawn from the plasmachain, and allow it to be transferred back to its owner's account on the mainchain.

Starting an exit for a coin requires providing the transaction that gave the exitor ownership of the coin signed by the previous owner in the coin's history, **tx**, as well as a direct

[14]Deposit blocks including only one transaction is an optimization, https://ethresear.ch/t/one-plasma-cash-block-per-deposit-why/2674/

Protocol for proving and verifying a coin's history

The prover sends two lists of `IncludedTx` elements to the Verifier, `inclTx` and `exclTx`. The verifier has access to the Merkle Roots from the deployed Plasma Contract on Ethereum via the variable `root[blockNumber]` as well as all the committed block numbers for the coin, via the variable `blocks`.

We consider a VerifyMerkleProof(slot, hash, proof, blockNumber) algorithm which is able to verify the merkle proof for a coin at a certain block number.

```
1: inclTx, exclTx ← proof
2: // Ensure completion of included and excluded transactions' blocks
3: Assert inclTx.keys ∪ exclTx.keys = blocks
4: // Ensure separation between blocks in included and excluded transactions
5: Assert inclTx.keys ∩ exclTx.keys = ∅
6: LastBlock ← DepositBlock
7: LastOwner ← DepositOwner
8: // Check the deposit transaction
9: if !VerifyMerkleProof(slot, inclTx[DepositBlock].tx.hash, itx[DepositBlock].proof,
   itx[DepositBlock].blkNumber) then
10:     return false
11: delete inclTx[DepositBlock]
12: // Check that included transactions are correct in the included blocks
13: for itx in inclTx do // Skip the deposit transaction
14:     if !VerifyMerkleProof(slot, itx.tx.hash, itx.proof, itx.blkNumber) then
15:         return false
16:     // Reject double spends
17:     if LastBlock ≠ itx.tx.parentBlock then
18:         return false
19:     // Accept spends only with valid signatures
20:     Sender ← ecrecover(itx.tx.hash, itx.tx.sig)
21:     if Sender ≠ LastOwner then
22:         return false
23:     LastBlock ← itx.blkNumber
24:     LastOwner ← itx.tx.newOwner
25: // Check that there are no transactions in the excluded blocks
26: for itx in exclTx do // itx.tx should be empty
27:     if !VerifyMerkleProof(slot, emptyHash, itx.proof, itx.blkNumber) then
28:         return false
```

Figure 3: Proving and verifying a coin's history in Plasma Cash

ancestor of that transaction (the reason the parent transaction must also be provided is explained in Section 4). Merkle proofs of inclusion need to also be provided for both transactions.

Each coin has internal state (s, c, T), where s indicates the exit phase (not exiting, exiting, exited), c the number of outstanding challenges, and T the time after the coin's exit can be finalized.

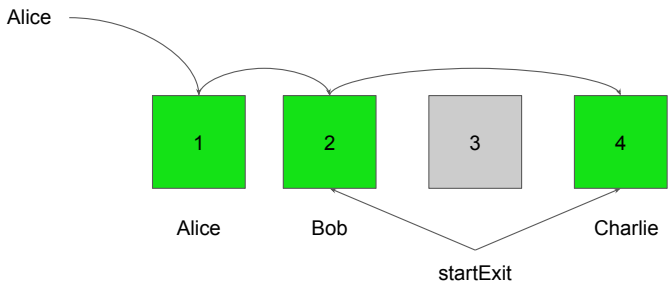


Figure 5: Alice deposits a coin from the mainchain to the plasmachain in Block 1. Alice sends the coin to Bob in Block 2. Bob verifies the inclusion of the coin in Block 2. Block 3 gets submitted, without including the coin. Bob sends the coin to Charlie in Block 4. Charlie has to verify the inclusion of the coin in Block 1 and 2, and the non-inclusion of the coin in Block 3. In order for Charlie to exit the coin received by Bob he has to provide the signed transaction from Bob as well as a direct ancestor, in this case the transaction from Alice to Bob. Charlie also needs to supply merkle proofs of inclusion for both of these transactions at their respective blocks.

A coin can be modelled by a state machine. After starting an exit, the coin transitions to the `EXITING` state. After the challenge (or maturity) period passes, the coin's exit can be finalized and it can transition to the `EXITED` state, from which it can be withdrawn to a user's wallet, as shown in Figure 6. Figure 7 illustrates the lifetime of an exit from its initialization to its finalization. We further discuss challenges in Section 4.

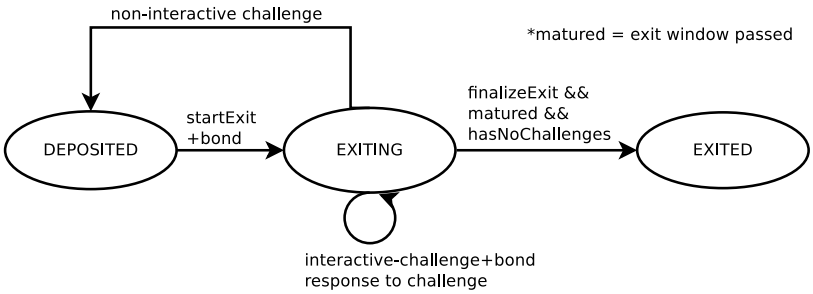
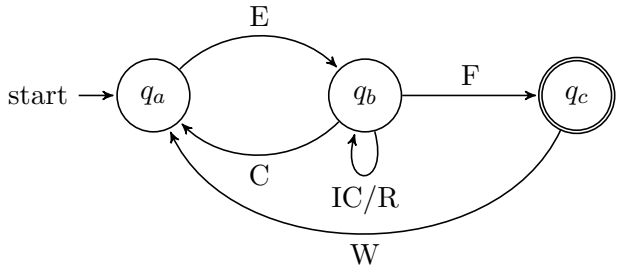


Figure 6: The stages of an exit. After a coin transitions to the `EXITED` state, it can be withdrawn to its owner's wallet.

A coin's state machine and the exit game.

Each coin has: (s, T, c) for its state, maturity period, and challenge count. A coin can only be withdrawn when it is in state q_c. T is a security parameter which represents the *challenge period*. *now* represents the current timestamp. Each included transaction (itxn) is of the form (blockNumber, proof, tx). itx.Sender is the address recovered from the signature on the hash of itx.tx.



1. StartExit (E) $(pitx, itx) : (q_a, 0, 0) \rightarrow (q_b, T, 0)$. Require $pitx.blockNumber < itx.blockNumber$. Store:
   (a) $eBlk : itx.blockNumber$
   (b) $epBlk : pitx.blockNumber$
   (c) $eOwner : itx.newOwner$
   (d) $epOwner : pitx.newOwner$
2. Challenge (C) $(itx) : (q_b, T, c) \rightarrow (q_a, 0, 0)$
   (a) SpentCoin *(challenge with a direct spend of the exiting transaction)*: $(itx.blockNumber > eBlk) \cap (itx.Sender = eOwner)$
   (b) DoubleSpend (*challenge with a direct spend of the parent transaction)*: $(epBlk < itx.blockNumber \leq eBlk) \cap (itx.Sender = epOwner)$
3. Interactive Challenge (IC) (itx): (*challenge with a transaction before the exit*). Require $(itx.blockNumber < epBlk) \cap (now < T/2) : (q_b, T, c) \rightarrow (q_b, T, c + 1)$. Store:
   (a) $cOwner : itx.newOwner$
   (b) $cBlk : itx.blockNumber$
4. Respond (R) (itx) : *(respond with a direct spend of the challenge transaction)*: $cBlk < itx.blockNumber \leq epBlk \cap itx.Sender = cOwner.(q_b, T, c) \rightarrow (q_b, T, c - 1)$
5. Finalize (F): Requires $now > T$. $(q_b, T, 0) \rightarrow (q_c, T, 0)$
6. Withdraw (W): $(q_c, T, 0) \rightarrow (q_a, 0, 0)$

Figure 4: The basic exit game for a coin

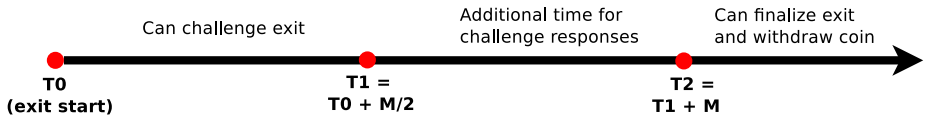


Figure 7: During its lifetime, an exit can be challenged during the maturity period. After its maturity period is over, it can be finalized and the exiting coin can be withdrawn.

4 Attacks on Plasma Cash

In this section we model the types of fraud attempts that users can engage in with or without collusion with the plasmachain's consensus mechanism. We create challenges that guard against malicious behavior, guaranteeing safety as long as the party being attacked logs in[15] at least once during the dispute period. The challenges described were first introduced in [2, 11] and are further explained in this paper for clarity.

4.1 Exit Spent Coin

This is an attack which does not require collusion of any of the transacting parties with the operator. An attacker (Alice) sends a coin to a victim (Bob). After both parties verify its inclusion, the attacker immediately attempts to exit the spent coin[16]. Bob must log in before the end of the challenge period and provide proof of a direct spend of the coin. An example of this is shown in Figure 8.

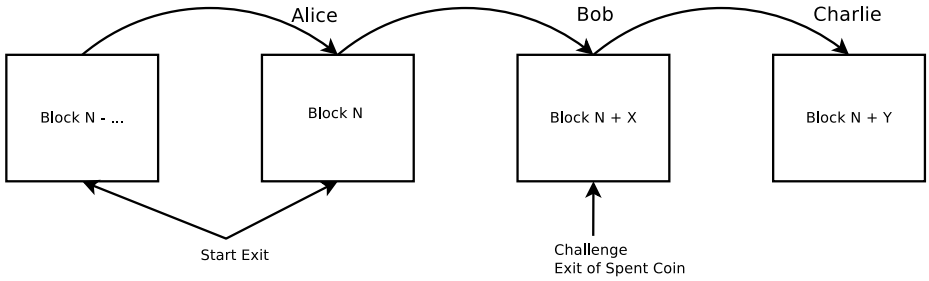


Figure 8: Alice attempts to exit the coin that was included in Block N. A valid challenge provides the inclusion of the transaction at Block $N + X$. Challenging with the transaction from Block $N + Y$ is not valid since it assumes the validity of its ancestors.

In the reference implementation, this is implemented via `challengeAfter`.

4.2 Exit of a Double Spend

This attack requires that the operator allows the inclusion of double spend transactions (ie. two transactions involving the same coin and ancestor transactions but different new

[15] Once the user logs in, they should challenge any exits that have been initiated for the coins they own

[16] Alice also supplies a security bond as discussed previously, to incentivize users to challenge if her exit is fraudulent, like in this case

owners included in two different blocks). In this case, The attacker (Alice) sends a coin to the victim (Bob). After the inclusion of the transaction to Bob, Alice sends another transaction to her colluding party (Charlie). Note that the operator here should notice that Alice no longer owns the coin and reject the transaction, however we consider that they are also colluding with Alice to steal Bob's coin. From the mainchain contract's point of view, both Bob and Charlie are valid owners of the coin.

Charlie can exit by providing the same parent transaction as the one Bob, given that they both received it from the same spend of Alice. A valid challenge involves a transaction between Charlie's exiting and parent block, proving the double spend, as shown in Figure 9. Users should additionally check the coin's history for double spends when receiving a coin. In the below example, Charlie can send the coin to another user, the verification of the merkle proofs will pass, however if they find transactions in the coin history with the same parent they should only accept the transaction coming from the earliest owner of the coin, since all the others are double spends.

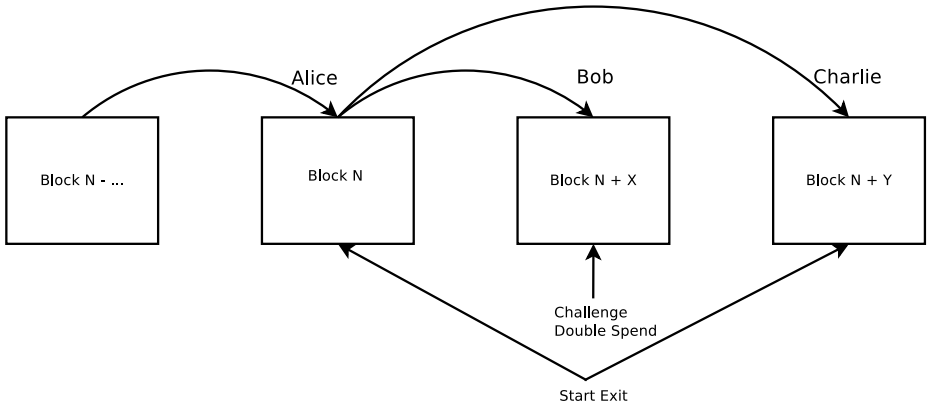


Figure 9: Charlie attempts to exit the coin that was included in Block $N + Y$, which is a double spend since the coin was already given to Bob at block $N + X$. A valid challenge provides the inclusion of the transaction at Block $N + X$.

In the reference implementation, this is implemented via `challengeBetween`.

4.3 Exit with Invalid History

This type of attack requires collusion of the transacting parties with the operator to include transactions which do not have valid ancestors. As an example, consider the case that the victim (Alice) has a coin that she does not intend to spend. The attacker (Bob) colludes with the operator to create a transaction that references an invalid transaction from Alice to Bob, effectively giving Bob ownership of the coin, and sends that transaction to his colluding party (Charlie). After the transaction gets included, the transaction to Charlie can be used as a valid ancestor transaction. Taking advantage of that, Charlie sends the coin to Dylan (another colluding party). From Dylan's perspective, he has a valid transaction signed from

the previous owner of the coin, along with a valid ancestor which is the transaction from Bob to Charlie. Dylan can now perform a seemingly valid exit.

Note that precisely because of this type of attack, we require that users must validate the full history of a coin they receive in a transaction. If Charlie was not a colluding party and they were a victim as well, he would not accept the transaction because he would have noticed that the coin has an invalid history.

When Dylan initiates the exit, Alice notices that and submits an interactive challenge, with which she claims that she is the latest valid owner of the coin. We require this challenge to be interactive and bonded, as it may be the case that Alice is not the last valid owner of the coin, and a spend of her to another user may exist. This leaves a window for a response to her challenge which invalidates it (contrary to the previous two challenges which were non-interactive). Multiple of these challenges can be active for an exit. During finalization the coin must have zero pending challenges that were not responded to, in order for the exit to be successful. The full game is shown in Figure 10.

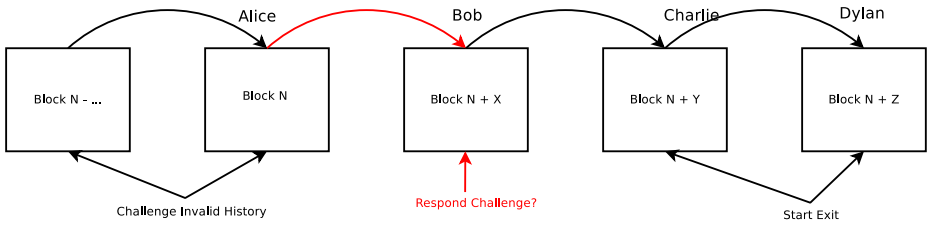


Figure 10: Dylan attempts to exit a coin that has invalid history. Challenger challenges by claiming that Alice is the last valid owner of the coin. A valid response must be a valid transaction that nullifies that claim. In this case this does not exist in this case since Bob and the Operator colluded to have the invalid transaction included. An Alice-to-Bob transaction could be used as a valid response as illustrated by the red arrows.

In the reference implementation, this is implemented via `challengeBefore` and `respondChallengeBefore`.

4.4 Withhold and Challenge Exit of Spent Coin (Griefing)

Griefing is a special type of attack which exploits inefficiencies of the protocol to 'bully' the transacting parties and either block them from some action (disallowing the exit of a coin for example) or forcing them to forfeit some constant amount of on-chain collateral when they attempt to settle a transaction on-chain.

In this attack, consider the scenario where Alice wants to send Bob a coin in exchange for a product. Bob will only send the product after he has validated the coin's history and after he has verified that the payment's transaction was included in a block. This requires that the operator makes the witness data for the transaction's inclusion available, so that both parties can validate the transaction's inclusion. Instead, the operator publishes a block including the transaction, however they withhold the witness data.

As a result, neither Alice nor Bob can know if the transaction was actually included. At this point, Alice must assume that the operator is malicious and initiate an exit for the coin (she can pay Bob through an on-chain transaction). The operator can now challenge the exit with a *Exit Spent Coin* challenge, as discussed in Section 4.1. During this process, they are required to reveal witness data (which they were previously withholding). This provides enough information for both parties to know that the transaction was successfully included in a block. However, in this process Alice incurred a constant griefing vector by losing the collateral she put up when initiating the exit as shown in Figure 11.

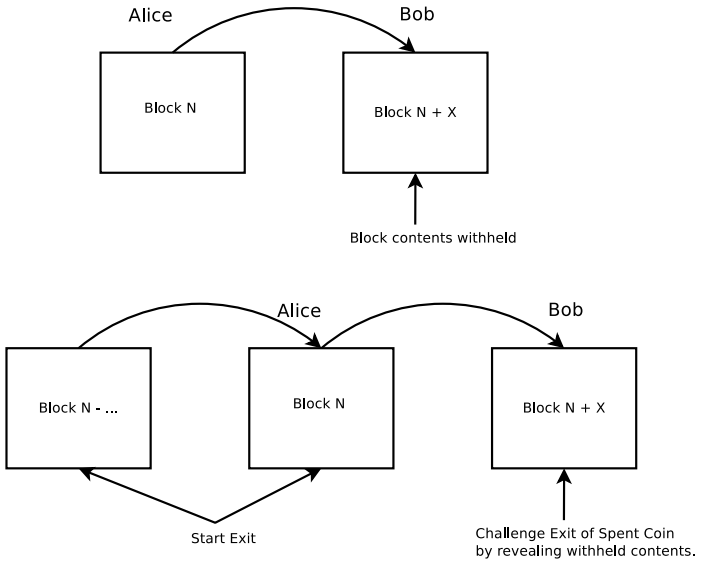


Figure 11: The operator steals the bond but the transaction was revealed.

This can be mitigated by constructing a different type of exit that forces the settlement of a withheld transaction to a receiving party other than the exitor. The coined term for this type of exit has been 'limbo exit' and the transaction being withheld is an 'in-flight' transaction, given that no party other than the operator can attest that the transaction has settled or not (and thus is in 'limbo' or still 'in-flight') [12].

5 Future Work

In this section we go summarize the existing proposals for improving the Plasma Cash protocol. In-depth analysis of each technique is not in scope and the reader is encouraged to refer to the citations for further insight.

5.1 Arbitrary Denomination Payments

The protocol for Plasma Cash is simplified compared to other Plasma versions due to the usage of non fungible tokens. This has the disadvantage that for financial use cases, such

as buying a product, a user has to provide a coin with the exact product value, or provide a coin with higher value and expect another coin as change, similar to how physical cash is used in real life.

Splits and Merges Proposals to allow coins to be split and merged are made in [13, 14]. Splitting a token involves creating new tokens that represent a previously unified token. Merging involves creating a new token out of a list of separate unspent tokens.

Plasma Debit A protocol that allows arbitrary denomination payments of a coin is Plasma Debit which acts as an extension to Plasma Cash [15]. In Plasma Debit, instead of having a set denomination, each coin has a capacity value v and a varying value a[17]. The following conditions apply:

1. after a coin is deposited $v = a$.
2. $a \leq v$
3. a of the coin's denomination is owned by the coin's owner
4. $v - a$ of the coin's denomination is owned by the operator

Each coin in Plasma Debit can be considered as a bidirectional payment channel between the Plasma Operator and the coin's owner. Arbitrary payments between two users can be initiated by reducing the balance of the sender's coin by an amount, and by atomically[18] increasing the balance of one of the receiver's coins by the same amount. The primary limitations in this construction is that the receiver must have a coin that is undercollateralized (e.g to receive X currency units, it must be the case that $X + a \leq v$).

Regarding the creation of undercollateralized coins, the protocol can be extended to allow the operator to deposit collateral, in order to generate coins with $a = 0$ on deposit, or increase the v while keeping a the same of an already existing coin. Markets can be expected to form where undercollateralized coins with larger v have more value than coins with smaller v, similarly to how payment channels with more capacity are more useful due to their increased routing capabilities.

Plasma Debit is a useful construct not only because it allows for arbitrary payments in Plasma Cash, but because it allows for behavior that can be characterized as transferrable-payment-channels. While traditional payment channels between two parties are non transferrable and require knowing all transacting parties beforehand, transferring a Plasma Debit coin from Alice to Bob is effectively the transfer of a payment channel between Alice and the operator to Bob.

[17] Each coin can be thought of as an account with maximum capacity of v

[18] Either all state transitions execute simultaneously, or none

Plasma Defragmentation When depositing a coin, users get multiple coins of very small equal value totalling the value of the deposited coin. Making a payment involves sending multiple of these small denomination coins to the receiver. By exiting a subtree of coins with the same owner, a user can efficiently withdraw and transact in arbitrary denominations. Users that utilize this technique eventually end up with various coins fragmented coins that are not efficiently exitable by a subtree. Defragmentation solves that by rearranging coins between the transacting parties so that users maintain lists with coins that have the most consecutive ids possible, allowing efficient subtree multi-exits.

5.2 Reduce user data checking

As discussed in Section 3.4, a receiver of a coin has to verify its history in order to make sure that the coin they are receiving is valid. Otherwise, they may be receiving a coin with invalid history, which if exited would be susceptible to *Challenge with Invalid History* from Section 4.3. A proof of inclusion or non-inclusion for a coin in a chain with 2^{16} coins is 512 bytes, per block. Assuming a Plasma Block frequency of 15 seconds[19], this means that after a year, a sender would be required to transfer more than 1 GB of data of proofs to the receiver, so that the receiver can validate the coin's history. This is expensive both in terms of bandwith and storage.

Checkpointing of coin history A solution to this problem is checkpointing the history of coins [16]. A plasmachain can be expected to have a large number of coins, and as a result a mechanism for efficient mass checkpointing of coins needs to be constructed. After a coin gets checkpointed, proving the validity of a coin's history requires checking the history of the coin from its latest checkpoint until the current block, which reduces the amount of merkle proofs that need to be validated.

Compressing non-inclusion proofs The largest part of data checking in Plasma Cash stems from the verification of non-inclusion proofs. ZK-SNARKs can be used to create a succinct constant-size proof of non-inclusion for multiple blocks for a coin which can be transmitted at low bandwith cost, and can be verified by the receiver in constant-time. Alternatively, a bloom filter hash can be published alongside the merkle root at each block by the operator, succinctly committing to the coin ids spent in each block, allowing users to verify the non-inclusion of a coin in blocks. This approach requires data availability from the plasma operator so that users can retrieve the raw bloom filter from its published hash, as well as revealing the bloom filter on-chain during an exit, which is expensive in storage terms. Finally, after an initialization phase involving a trapdoor parameter, an RSA accumulator can be published alongside the merkle root at each block by the operator, which allows batch proofs of non-inclusion of a coin, effectively reducing the amount of proofs required

[19]The Ethereum mainnet block time

for the validation of a coin to two (one for its inclusion in a block, and an aggregate proof for all the non-inclusions) [17].

5.3 Faster and Inexpensive Withdrawals

Tokenized Exits A user needs to wait a predefined challenge period when initiating an exit, in order to allow other users to verify that the exit is valid. User experience can be improved by allowing the exitor to instantly redeem the value of their exit by tokenizing it as a NFT and selling it in an open marketplace. The buyer of the tokenized exit will wait the challenge period instead of the exitor [18]. The tokenized exit's buyout price is equal to the value of the coin that is under the exit. The tokenized exit can be expected to be sold at a discount, depending on the exitor's time preference.

Optimistic Exits By assuming that a user is honest, the merkle proofs and the raw transactions can be omitted during the initiation of an exit. This reduces the exit's required transaction fees. An additional non-interactive challenge must be provided which reveals the previously omitted exit parameters and if the challenge is successful cancels the exit [19].

6 Conclusion

In this paper we presented Plasma Cash, a plasma construction which can be used to construct non-custodial arbitrarily centralized sidechains which maintain safety, contrary to two-way pegged sidechains which do not allow the safe withdrawal of funds in case the sidechain consensus is compromised. Plasma Cash can be used to offload computation from the mainchain and improve scalability, which is a pressing issue for decentralized permissionless blockchains. We presented the required mechanisms as well as examples of how users interact with the system in order to deposit, transact and withdraw their funds. A reference implementation which is used in production is provided. Further improvements to the protocol are described, with respect to allowing arbitrary denomination payments, improving user experience, and making light clients more efficient.

References

[1] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timn, and Pieter Wuille. Enabling Blockchain Innovations with Pegged Sidechains. `https://blockstream.com/sidechains.pdf`, Oct 2014.

[2] Vitalik 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`.

[3] Joseph Poon and Vitalik Buterin. Plasma: Scalable Autonomous Smart Contracts. https://plasma.io/.

[4] Plasma Implementers Call. https://www.youtube.com/channel/UCG2MeKuKDJRK4gFNk-dQuZQ.

[5] Kiayias Aggelos, Andrew Miller, and Dionysis Zindros. Non-Interactive Proofs of Proof-of-Work. https://eprint.iacr.org/2017/963.pdf, May 2018.

[6] Patrick McCorry, Chris Buckland, Surya Bakshi, Karl Wust, and Andrew Miller. You sank my battleship! A case study to evaluate state channels as a scaling solution for cryptocurrencies. https://nms.kcl.ac.uk/patrick.mccorry/battleship.pdf, Oct 2018.

[7] Jeff Coleman, Liam Horne, and Li Xuanji. Counterfactual: Generalized state channels. https://l4.ventures/papers/statechannels.pdf.

[8] Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. Efficient sparse merkle trees - caching strategies and secure (non-)membership proofs. *IACR Cryptology ePrint Archive*, 2016:683, 2016.

[9] Whats a Sparse Merkle Tree? https://medium.com/@kelvinfichter/whats-a-sparse-merkle-tree-acda70aeb837.

[10] Plasma cash with sparse merkle trees, bloom filters, and probabilistic transfers. https://ethresear.ch/t/plasma-cash-with-sparse-merkle-trees-bloom-filters-and-probabilistic-transfers/2006.

[11] Plasma cash simple spec. https://karl.tech/plasma-cash-simple-spec/.

[12] Li, Xuanji. Limbo Exits and Challenging Fraudulent Exits. https://ethresear.ch/t/limbo-exits-and-challenging-fraudulent-exits/2015.

[13] One proposal for plasma cash with coin splitting and merging. https://ethresear.ch/t/one-proposal-for-plasma-cash-with-coin-splitting-and-merging/1447.

[14] Dan robinson on splitting and merging in plasma cash discussion. https://ethresear.ch/t/plasma-cash-plasma-with-much-less-per-user-data-checking/1298/53.

[15] Plasma debit: Arbitrary-denomination payments in plasma cash. https://ethresear.ch/t/plasma-debit-arbitrary-denomination-payments-in-plasma-cash/2198.

[16] Plasma xt: Plasma cash with much less per-user data checking. https://ethresear.ch/t/plasma-xt-plasma-cash-with-much-less-per-user-data-checking/1926/.

[17] Buterin, Vitalik. RSA Accumulators for Plasma Cash history reduction. https://ethresear.ch/t/rsa-accumulators-for-plasma-cash-history-reduction/3739/.

[18] Kelvin Fichter. Simple Fast Withdrawals. https://ethresear.ch/t/simple-fast-withdrawals/2128/1.

[19] Buterin, Vitalik. Optimistic cheap multi-exit for Plasma (Cash or MVP). https://ethresear.ch/t/optimistic-cheap-multi-exit-for-plasma-cash-or-mvp/1893.