Plasma XT embraces Plasma Cash's original vision of simple, reliable, low-cost transactions for everyone in the world.*

Special thanks to Dan Robinson for discussion and coming up with much of this, as well as David Knott, Joseph Poon, Karl Floersch, and Vitalik Buterin for ideas and feedback that led to this design. Another special thanks to Justin Drake for the gorgeous construction of cryptoeconomic aggregate signatures and to Sunny Aggarwal for the Plasma XT name.

(enough memes now)

Really, huge thank you to @danrobinson! Almost all of this came out of discussion with/ideas from Dan.

*context

TL;DR

Plasma XT is a modification to Plasma Cash that enables safe checkpointing.

Background

Plasma Cash, as originally specified, requires that users maintain an ever-growing history of transactions, inclusion proofs, and non-inclusion proofs. Currently, this history is too large to be feasible.

Some napkin math to illustrate:

Assuming we have a (very) small Plasma Cash chain with only 2^16 (= 65536) coins, then the Merkle proof of either existence or non-existence for a given coin will be at least 32 bytes * 16 siblings = 512 bytes per block. If we assume that Plasma block is created once every 15 seconds, then we'll end up with 31557600 / 15 = 2103840 blocks per year. 2103840 blocks per year * 512 bytes per block = 1077166080 bytes per year = ~1.077 gigabytes per year. 1 gigabyte per coin, per year!

This means that the owner of a coin would, after one year, have to send more than a gigabyte of proof data simply to make a single transaction. Although the exact proof size can likely be slightly optimized, the proof size still grows linearly with the size of the Plasma Cash chain. That's a lot of data! It'd be hard to store and validate a proof this big on devices that might be computationally limited, like phones or embedded devices. After enough time, it might even be difficult to validate these proofs on laptops or workstations.

This post attempts to solve this problem by specifying a protocol in which coins can be checkpointed. The owner of a coin will, therefore, only need to provide a proof back to the coin's last checkpoint.

Checkpoints

Checkpoints are one solution to the above problem. If we can properly construct regular intervals in which the state of a coin is considered "finalized", then it's possible to reduce the total proof size significantly (to some relatively small constant). Generally, this is accomplished by having the operator attest to the current state and allow for a challenge period in which this state may be contested. If no one shows that the state is invalid, then the checkpoint is considered finalized.

Naive Checkpoints

Checkpoints are really hard to get right! Let's look at the following (flawed) checkpoint construction to illustrate some of the challenges:

Every N

blocks, the operator publishes a Merkle state root. This root is derived from a Merkle tree in which each leaf node represents the current valid owner of each coin in the tree. So if user A

owns coin #0

, then the operator will place A

's address at the 0th (first) leaf in the tree.

The operator places a bond on this action. Any user can prove that the state is invalid (and therefore claim the operator's bond) by showing that the operator has lied about the owner of a certain coin. If no successful challenges take place after some amount of time, then the checkpoint is considered finalized.

Why is this design flawed? It comes down to the (really hard) problem of data availability. If the operator publishes a state root but does not publish the tree itself, then it's impossible for anyone to challenge the checkpoint. You can't prove that something is invalid if you can't even validate it in the first place.

Because we don’t know if the checkpoint is valid or not, we have to assume that the operator is trying to steal some money. Once the checkpoint finalizes, the checkpoint state becomes the “true” state. The only solution is for all of the coin owners to exit before the checkpoint finalizes

. Even worse, since we can’t prove the operator did something bad, we can’t punish the operator. We never want to force a user to act within some time-frame if the user won’t receive some sort of bounty for doing so.

Fancy Checkpoints

It turns out we can (sort of) solve the data availability problem by taking advantage of a few interesting observations. Let’s go back to first principles - in the above example, no one can challenge the operator because no one knows if the data is valid or not. So, what if we could design a system so that someone will

know if the operator is trying to cheat?

Well, we can! We just need to rely on some signatures. The basic idea here is that each user that wants to have their coin checkpointed will sign off on their coin in the checkpoint

. Generally, this means that the operator has proof

that each coin holder that wants their coin checkpointed has confirmed the checkpoint to be correct. Then, if the operator claims to have your permission but doesn’t, you know the operator is trying to cheat and the operator can be punished.

Cryptoeconomic Aggregate Signatures

But wait, wouldn’t it be too expensive to put all those signatures on-chain? Yes, absolutely. But luckily, we don’t actually have to put all of these signatures on-chain. Here’s where cryptoeconomic aggregate signatures come into play.

The idea here is that instead of publishing full signatures for every user, the operator only publishes a claim

that a certain owner has provided a signature (in the form of a bitfield). For example, if we have 4 coins, and owner of coins #0

and #3

have provided signatures, then the operator would publish the bitfield “1001”. If, however, the owner of coin #3

did not

provide a signature and the operator published that bitfield, then the owner of coin #3

could challenge by requesting the operator provide the owner’s signature. So our signatures are suddenly down to one bit per coin!

Checkpoint Zones

But wait, wouldn’t one bit per coin still

be too expensive to put on-chain? Well, yes. Our scheme just requires putting the bitfield in calldata, so things work out to about 100k gas (currently ~$0.25) per 8192 coins = ~1 kilobyte. This is too much gas if we want to checkpoint many coins (millions) at the same time, so we turn to “checkpoint zones”.

Checkpoint zones effectively checkpoint some

coins at a time. Every coin will be scheduled to be checkpointed on some regular basis. For example, coins 0-8191 might be checkpointed during one block, and coins 8192-16383 might be checkpointed in the next. By checkpointing on a rolling basis, we make sure that we’re always inside of the gas limit, even if we need to checkpoint millions of coins.

Challenging a Checkpoint

Users can challenge invalid checkpoints. Checkpoints can be invalid if the operator places a 1 at the index of a certain coin but the true owner hasn’t signed off on that checkpoint. Only the true owner knows if they’ve signed off or not, so each coin owner is responsible for verifying the bitfield in each checkpoint. Luckily, the user is already required to be online regularly to look for invalid exits on their coins, so we’re not changing any assumptions behind Plasma Cash.

If the operator places a 1 in the bitfield for a user that did not sign off on a signature, then the user will challenge. This challenge includes the user’s latest transaction in the history, attesting that the user is indeed the owner. The operator can then respond by either showing a signature from the owner or with a later transaction proving that the challenger is not really the owner. Multiple challenges may exist on the same coin at the same time, but only one bond will be paid per coin.

There's a potential attack vector if the operator submits very many 1s in the bitfield at the same time, because each user that didn't sign off will have to submit a challenge. To get around this, we define a maximum number of challenges that may be open per checkpoint at any one time (probably 256). If more than this max number of challenges are open at any one time, then the checkpoint is invalidated. This basically ensures that not everyone

needs to challenge and we don't overload the root chain with challenges.

Caveats

Gas Cost

As we stated before, the bitfield grows with the total number of coins in the Plasma Cash chain, so things get more expensive as the total number of coins increases. This checkpoint scheme is a good solution for even many millions of coins, but might need to be revisited if we ever reach many tens of millions or even hundreds of millions of coins.

Liveness

We're assuming that a user will be live at least once during the checkpoint challenge period. This doesn't change any assumptions of Plasma Cash if we make the checkpoint challenge period at least as long as the exit challenge period.

Griefing

By Operator

The operator can grief a user by refusing to checkpoint a coin (always putting 0 in the bitfield). If there's a 0 at some index, it's impossible to tell if the operator didn't receive a signature or is simply refusing to include it. This doesn't harm any security, but means that we're falling back on Plasma Cash-sized transaction history lengths. We could probably implement some sort of manual checkpointing as an on-chain transaction, but we want to avoid that if possible.

By Users

Users can grief other users by making lots of open challenges and invalidating the checkpoint. If the open challenges aren't valid, then this attack is very costly as each challenge requires a bond. This also doesn't harm security, but might be annoying. We need to determine the optimal bond size that makes this attack as costly as possible without making it too costly to submit a challenge. Again, this case simply means that we fallback on the guarantees and security properties of Plasma Cash.