Scalable Privacy

Daira Hopwood
<daira@electriccoin.co>
@feministPLT
@daira on chat.zcashcommunity.com
Slides at https://github.com/daira/zcon

Disclaimers

- This is an opinionated talk that covers only areas of the solution space that I consider most promising.

- The Electric Coin Company has committed to researching scaling, and aims to deploy a solution by 2021. It has not committed to using any of the specific techniques described here.

- Zcash Sapling has pretty good performance, in concrete terms, already.

Problem statement

- A protocol is "*horizontally scalable*" if by adding compute power in parallel, it is possible to support high (not unlimited) transaction throughput that scales roughly with the number of nodes, and a large number of users.
- We don't demand perfection – in any practical protocol there will be bottlenecks. I still consider a protocol to be horizontally scalable "in practice" if those bottlenecks only manifest at scales much higher than anticipated usage.

Deploy a version of Zcash that is horizontally scalable in practice without compromising on privacy.

Problem statement

- A “version of Zcash” does not have to be compatible with every aspect of current Zcash.
  - It could be an entirely new protocol and block chain, provided that value can be transferred from the old protocol, and the combined system remains usable for the transition.
- “Without compromising on privacy” implies that:
  - Amounts, senders, and receivers remain private (i.e. the transaction graph is private).
  - The note traceability set of any input is “all” previous outputs (that the adversary cannot rule out by information independent of the block chain).

Secondary goals

- Allow light clients with weak trust requirements.
  - A light client can fully verify the block chain with low bandwidth and storage.
- Improve privacy by use of network-layer privacy mechanisms.
  - Current Zcash has excellent on-chain privacy but sends transactions in the clear.
- Reduce cryptographic assumptions.
  - The fact that the zk-SNARK parameter setup requires trust, is a big issue for confidence.
- Solve transaction malleability.
  - A good payment protocol should be able to provide certainty to payer and payee that a transaction occurred; malleability interferes with this.

Non-goals

- Does not need to support general computation.
- Does not need to be a direct upgrade of the existing Zcash peer-to-peer network.
  - There are several possible ways to preserve continuity of the ZEC token without that.
  - We need to be able to verify that scaling actually works on a testnet, first.
- Does not need to support transparent addresses or transactions.
  - It’s prohibitively hard to verify Bitcoin script in an R1CS circuit.
  - If we can’t verify Bitcoin script, we don’t get Bitcoin compatibility anyway.
  - With Sapling (even more so with circuit-friendly hashes such as Rescue or Poseidon), shielded proving time is entirely practical.
  - Transparent / only partially shielded transactions are a significant privacy weakness.
- Does not need to reuse the Sapling protocol unchanged.

Why not general computation?

- A private, scalable, censorship-resistant currency is hard enough.
  - We have to explicitly design a protocol that is private and scalable.
  - This is not easy to do by "adding privacy" to a scalability design, or by "adding scalability" to a privacy design.
- The design of private scalability described in this talk can be combined (at significant effort) with other proposals – Zexe, etc.
- Good, auditable contract languages are an unsolved research problem.

Out of scope *for this talk*

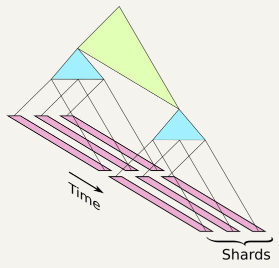
- Reducing latency relative to existing block chains.
  - The reduction in verification work per block will probably allow a latency improvement, but other necessary changes may increase latency again.
- Secure node discovery.
- Alternative designs for assigning resources to shards.
- Details of incentivization for miners and various helpers.
- Solving Proof of Work’s security problems or resource usage.
- Exactly how to preserve continuity from the existing ZEC token.
- These are all important problems, but mostly orthogonal to the techniques in this talk.
- If you’d like to talk about interactions of these with scalability and privacy, come to my and Nathan’s workshop this afternoon!

Privacy properties

- Amounts are “easy” to keep private.
  - It suffices to use homomorphic value commitments and range proofs.
- But this is not enough.
  - In a UTXO-style currency, the note traceability set of an input is the set of outputs that could be consumed by that input.
  - There are devastating privacy leaks in systems with small note traceability sets, especially (but not only) under active targeted attack.
  - “Plausible deniability” is not good enough in the real world.
  - Watch Ian Mier’s talk “Satoshi Has No Clothes” for details.
  - These attacks become infeasible if an input could correspond to “any” previous output (roughly speaking).

Proposal

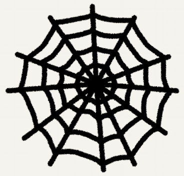
- Sharding, to scale to high transaction volumes.



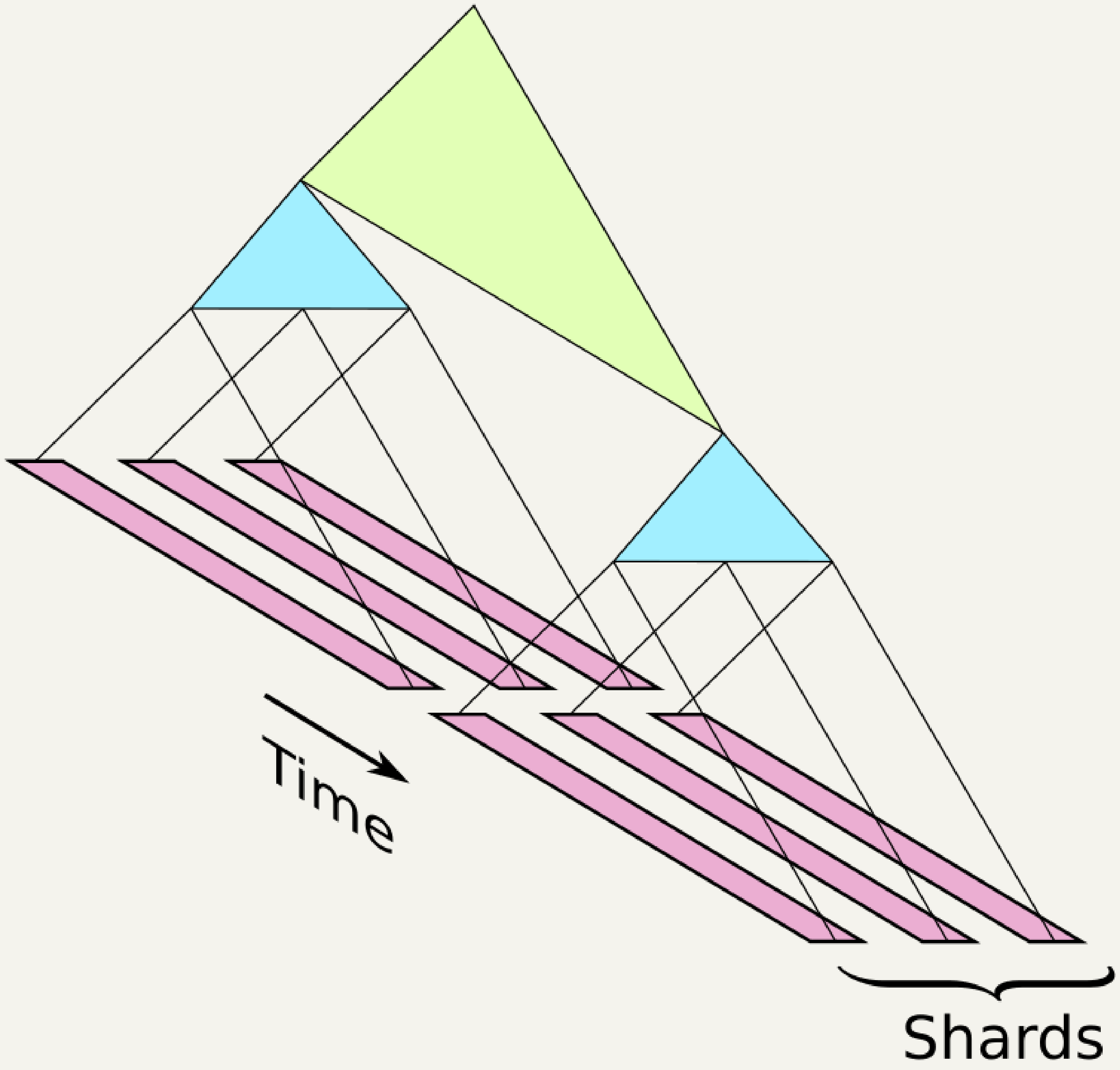
- Recursive SNARK validation, to create summaries of state updates and to allow clients to catch up almost instantly.

- SHARKs, to limit reliance on trust assumptions.

- A mix-net, to replace broadcast of transactions to miners and all potential token receivers, while maintaining privacy.



Sharding + recursive validation



Block chains

- Let’s review the block chain model (as used in current Zcash).
- We have transactions that describe changes to the system state.
- Sets of transactions are collected into blocks.
- A chain is a sequence of time-ordered blocks.
- We have a set of “*consensus rules*” that determine which block chains are *valid*.
  - Most transaction consensus rules depend only on a single transaction and the *previous* block (“*independent rules*”).
  - The only *essential* exception is the rule preventing *concurrent* double spends.
- We have some consensus protocol that results in nodes eventually agreeing on a prefix of some valid chain.
  - This requires certain assumptions that we won’t go into, such as a node having an adequate network view, and an adversary having a bounded proportion of the hash power (in PoW systems) of the whole network.
- In more detail: we actually have a *tree* of possible valid chains, and some rule for deciding which is the “best” valid leaf.

How Zerocash/Zcash works

- At a high level, we use the UTXO model from Bitcoin.
- Carriers of value are called “*notes*”.
- Notes have a value, a “*commitment*”, and a “*nullifier*”, and are linked to an address.
- Commitments are binding (only one note can correspond to a commitment).
- Commitments are hiding (they don’t reveal information about the note).
- Creating an output note:
  - reveals its commitment, which goes into a commitment tree;
  - fixes a unique nullifier, which is not revealed.
- Spending a note:
  - reveals its nullifier, preventing it from being spent again;
  - proves (in zero knowledge) that it was a valid note, by giving its path in the commitment tree;
  - proves (in zero knowledge) that the nullifier and commitment are correctly linked;
  - demonstrates spend authority for the note’s address.
- The state of the commitment tree and nullifier set is called a *treestate*.

Zero knowledge proving systems

- A statement is a proposition we want to prove. It depends on:
  - Instance variables, which are public.
  - Witness variables, which are private.
- Given the instance variables, we can find a short proof that we know witness variables that make the statement true, **without revealing any other information**.
- In some systems, proofs are constant-size and can be verified in constant time.
- A proof of knowledge is stronger and more useful than just proving the statement is true. For instance, it allows me to **prove that I know a secret key**, rather than just that it exists.
- **The proof can be just a string; anyone can verify it without interacting with the prover.**
- I'm glossing over some details, such as setup and variations of the security properties.

How Zerocash/Zcash works, contd.

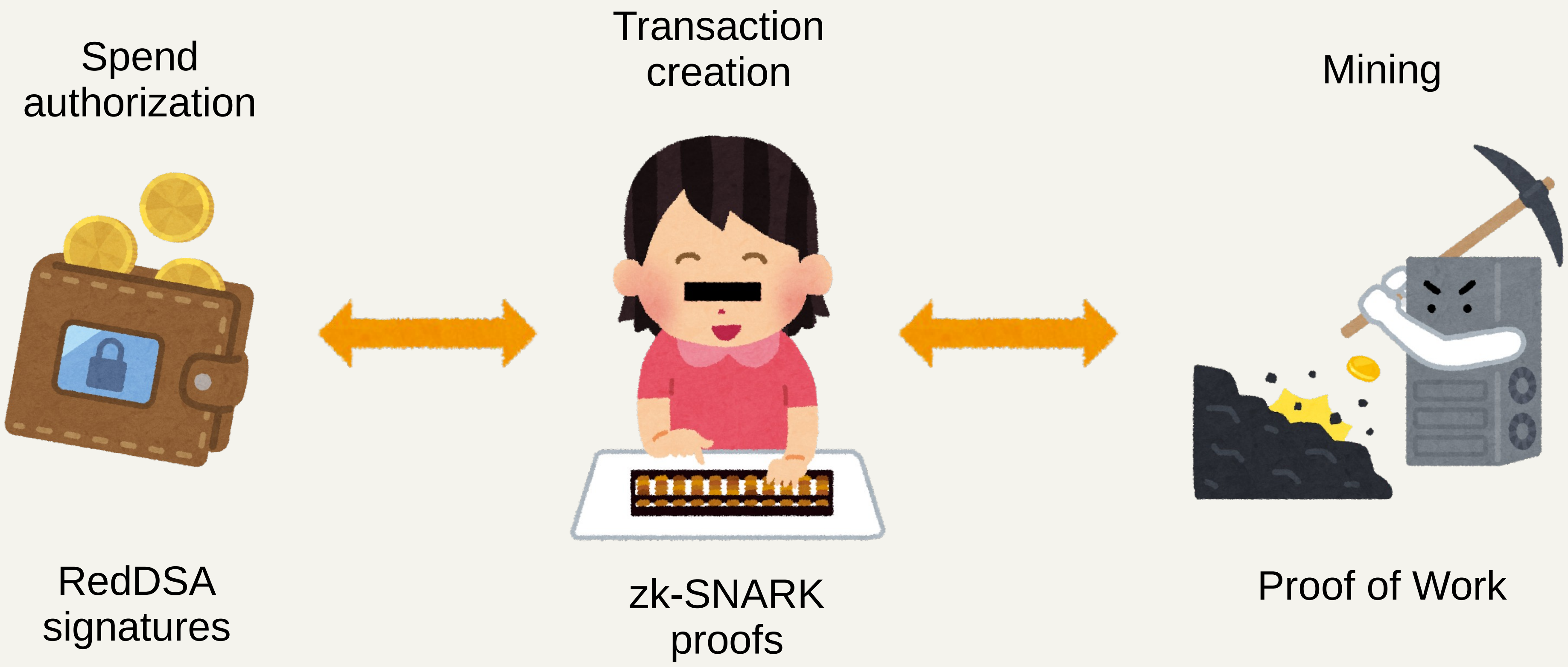
- How does a token-receiver know which transactions are sending tokens to it?
- Addresses also have *incoming viewing keys*.
- Output descriptions are public-key encrypted to the key in the destination address.
  - The encryption is *key-private*, so that the ciphertext does not reveal information about the key/address.
- Each token-receiver trial-decrypts *all* transactions using its incoming viewing key.
  - We are treating the block chain as a broadcast channel for all transactions.
- Privacy doesn’t depend on soundness of the proof system.

Sapling-specific detail

- Transactions have a Spend proof for each shielded input, and an Output proof for each shielded output.
  - Spend proofs are essential to the idea of the protocol: we need to prove in zero knowledge a rather complicated statement involving showing a path in the commitment tree, deriving the nullifier via a PRF, etc. This requires the full power of zk-SNARKs. Output proofs are more of a technical convenience.
- Spend and Output proofs don't directly check that values balance; they do so via homomorphic value commitments (like in Confidential Transactions, Monero, or Mimblewimble).
- The proving system makes use of a particular scalar field efficient in the circuit. We rely on that heavily for optimization, using Pedersen hashes in an elliptic curve over that field, etc.
- Viewing keys and payment disclosures allow selectively revealing the content of transactions.
- Other features (diversified addresses, possibility of threshold shielded multisig) don't interact with scaling.

Separated signing

- Spend authorization is demonstrated by a signature (with randomized public key for privacy), so that it can be done on a hardware wallet.



Protocol bottlenecks

- All validators must validate each transaction.
- All validators must update the commitment tree.
- All validators must update the nullifier set.
- All transactions must be broadcast to every validator and every potential token-receiver.
- All token-receivers must check each output in case it is sent to them.
- Miners must validate transactions for inclusion in blocks.
- Blocks must be broadcast.
  - A block includes its transactions, each of which must have been broadcast at some point.
- Validators that have been off-line must catch up to the current head.

Scaling

- Conventional block chains aren’t scalable.
- Every full validator needs to check every transaction.
- Miners are supposed to check every transaction… but we have no reason to trust miners.
- What if a miner included a proof that the transactions were valid?
- Problems:
  - There are a variable number of transactions in each block.
  - Doesn’t address bandwidth issues.
  - Doesn’t address detection of transactions involving you.
  - Doesn’t address storage of commitment trees and nullifier sets.
- Broadcast communication of all transactions for anonymity also isn’t scalable.
- So we need to combine this with other ideas.

Sharding

- *Sharding* is a common technique for achieving horizontal scalability.
- The basic idea is that we split *some aspect of* a distributed system into multiple communicating copies, or shards.
- Sharding can be more or less *obtrusive* at the payment system layer.
  - If it is obtrusive, then *payment system layer* resources may belong to particular shards, and there may be greater overhead for cross-shard interactions than intra-shard.
  - If it is **user**-obtrusive, then this resource allocation/overhead is visible to users. If it is **wallet**-obtrusive, then it is visible to wallet (and potentially other ecosystem) software but hidden from the user.
  - If it is unobtrusive, then *payment system layer* resources are global and there is uniform overhead.
- The approach described in this talk is wallet-obtrusive. (Notes are assigned to shards; addresses are not.)
- I believe user-obtrusive sharding is unnecessary.

Clarifications about sharding

- It’s not necessary for shards to be complete copies of the system.
- We can pick and choose which resources to shard based on where the bottlenecks are.
- The sharding design I’ll be talking about:
  - does **not** shard addresses (there is one global address space);
  - does **not** split the anonymity set;
  - does **not** require end-users to be aware of sharding (although it can affect latency);
  - *does* shard processing of transactions;
  - *does* shard storage of the archival block chain;
  - *does* shard processing of nullifiers;
  - *does* shard the set of notes.
- A client sending a transaction to a shard doesn’t need to “catch up” with other transactions.
- Technically, there is an limit on scaling due to communication between the shards. But this is not a practical obstacle to achieving the scaling we want.

Censorship-resistant consensus

- The hard part of distributed consensus is agreeing on the input.
- Three techniques:
  - Competition
    - e.g. PoW;
  - Sortition (selecting parties at random)
    - most PoS protocols have this as a component;
  - Fault tolerance
    - e.g. BFT protocols, MPC, anytrust.
- Consensus protocols often combine these.
- For this talk, assume we have a censorship-resistant way to select non-conflicting transactions for each time period *in each shard*.

Zero knowledge proving systems

- A statement is a proposition we want to prove. It depends on:
  - Instance variables, which are public.
  - Witness variables, which are private.
- Given the instance variables, we can find a **short** proof that we know witness variables that make the statement true, without revealing any other information.
- **In some systems, proofs are constant-size and can be verified in constant time.**
- A proof of knowledge is stronger and more useful than just proving the statement is true. For instance, it allows me to prove that I know a secret key, rather than just that it exists.
- **The proof can be just a string; anyone can verify it without interacting with the prover.**
- I’m glossing over some details, such as setup and variations of the security properties.

Recursive SNARKs

- A conventional SNARK has a fixed-size statement.
- We can work around this by using a tree of proofs.
- Because proofs can be verified in constant time, the verification can be expressed in a statement. Each non-leaf node proves that “I know n verified child proofs”.
- We end up proving all of the statements at the leaves of the tree.
  - We don’t prove that any single party knew all of the witnesses at the leaves. This is a feature, because we don’t have any single party that knows all of the private keys used in transactions.

Recursive SNARKs

- In general we can transform protocols that use zk proofs in a “flat” way, where all proofs are public, into protocols that use trees of proofs as necessary.
  - However, we might need to add proofs of things that were previously proven outside zk-SNARK circuits, in order to make this work.
- The proofs at each layer can use different proving systems or parameters.
- Security properties we need from the proving system:
  - All SNARKs, even if public-coin, have a trapdoor that allows creating valid proofs of false statements (or statements for which the witness is unknown). Therefore we need knowledge soundness, since it is vacuous that a valid proof *exists* for any instance.
  - We don’t necessarily need *zero knowledge* for scaling.
- Aside for language geeks: we can model protocols that use recursive proofs using S-attributed grammars.
  - A string in the language is the whole state we’re proving things about; the synthesized attributes are summaries.

Drawbacks of SNARKs

- "Private-coin" SNARKs require a trusted parameter setup. If the setup goes wrong, then proofs can be forged.
- The design of parameter setup protocols has advanced considerably. We now know how to create proving systems (e.g. Sonic) with updatable parameters and "universal" setups that work for any statement.
- But, pairing-based SNARKs are subtle, and it would be nice to base security on weaker (discrete log) assumptions, with *no* trusted parameter setup.
- Systems such as Bulletproofs satisfy this, but:
  - Verification time is roughly linear in the circuit size (and arguably too long in practice).
  - Proof size is roughly logarithmic in the circuit size, not constant.

SHARKs

- SHARKs are a new idea (and optimized proof system) due to Madars Virza, Mariana Raykova, and Eran Tromer.
- At a high level, a SHARK proof is:
  - An inner proof in a public-coin system (typically discrete-log-based).
  - An outer proof in a system with fast verification (typically pairing-based), stating that the inner proof is valid.
- This is similar to recursive validation, but only for one level, and we co-design the SHARK proving systems to make things more efficient.
- If the outer proving system breaks, you can recover by recreating the outer proofs, without knowing the inner witnesses.
- We only rely on the inner proving system of the SHARK for privacy.

SHARKs + Recursion

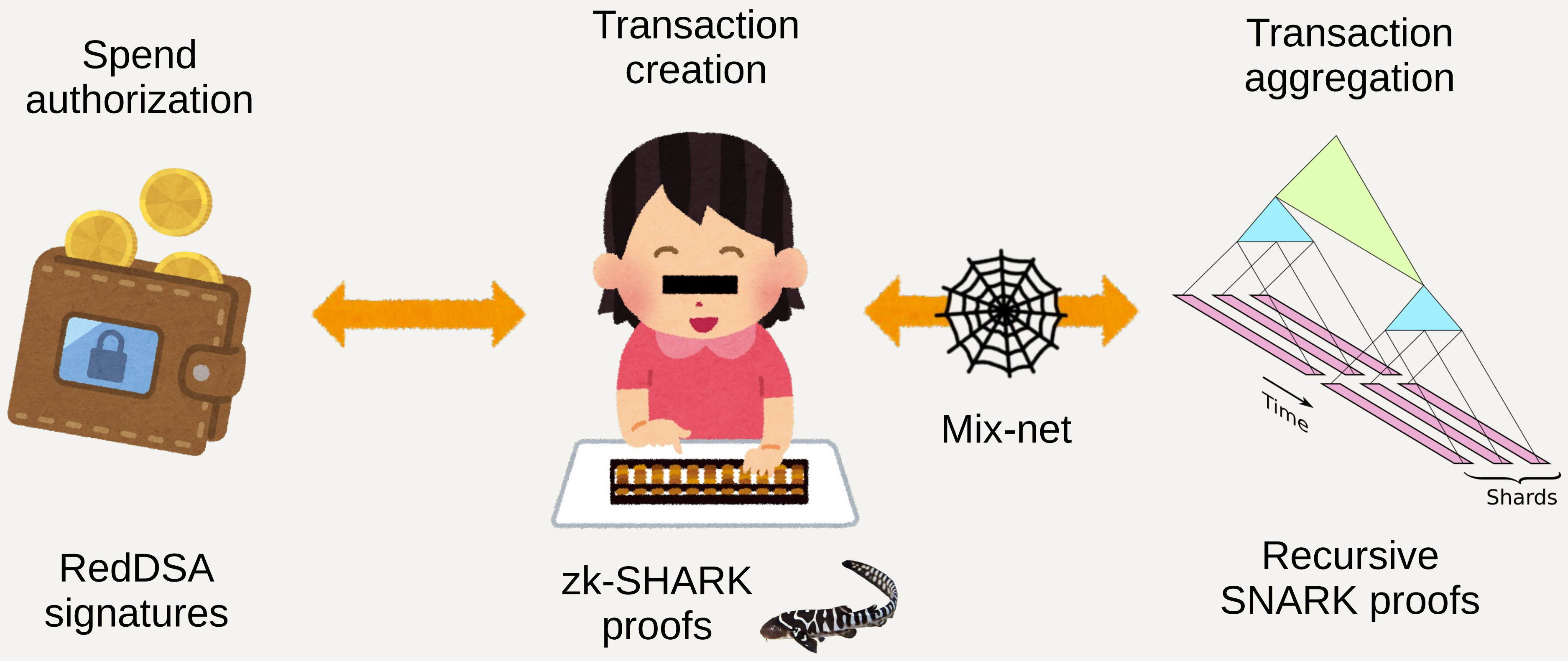
- Now consider combining SHARKs with recursive validation. We **must** have a validation procedure that is practical to express as a circuit at the next-higher level.
  - This is why we can't use, say, Bulletproofs for recursive validation. Each level would increase the validation effort, defeating the point of recursion.
- But, we can use a SHARK for leaf proofs, and fully succinct SNARKs for non-leaf proofs.
- Does this maintain the recovery property? Yes!
  - Provided that we archive all the inner SHARK proofs and instances, we can regenerate everything (at high cost), and be sure that the new summaries are consistent with the archive.
- *Both* the SNARK and the outer proving system of the SHARK can be completely broken, and we can still recover.
- We still only rely on the inner proving system of the SHARK for privacy (as far as proofs are concerned).

Efficiency of recursive SNARKs

- It turns out that the batch verification circuit of, say, Groth16 is quite efficient.
  - My current estimate is ~21000 R1CS constraints for the first proof and ~4700 constraints for each subsequent proof. Note that the Sapling Spend circuit is about ~96000 constraints.
  - This requires *lots* of optimizations and will need extensive auditing. It’s roughly as complicated as Sapling was (but we have more experience now).
- There’s a catch: verification is only efficient for “nested” curves, i.e. the scalar field of the outer proving system must be the field used by the inner proving system.
- The known ways of constructing these produce quite large (but not impractically large) curves.

Preserving separated signing

- Sending transactions to a shard via a mix-net, instead of broadcasting directly to miners, does not by itself affect separated signing.
- But, we have to be able to verify the RedDSA signatures in an R1CS circuit. This turns out to be quite practical.



Efficiency for transaction creators

- We want creating Spend proofs to still be practical on small client devices.
- So, we need to make sure the Spend circuit is still as small as possible.
- When using SHARKs, the transaction creator only strictly needs to create the inner SHARK proofs – one per input and output.
- The outer SHARK proof(s), and additional proof(s) for validation that would be done outside the circuit in Sapling, can *either* be done by the transaction creator or by a shard participant.
- Additional things that need to be proven include:
  - verifying RedDSA signatures;
  - verifying overall balance and nonmalleability;
  - nullifier set updates.
- Recursive validation proofs need to be done by a shard participant.

Accumulators and witnesses

- A SNARK statement must be represented by a fixed-size circuit.
- To prove things about large data structures, we can use cryptographic accumulators.
- These are protocols that allow acting on a potentially large structure via a succinct “summary”, and “witnesses” for particular elements.
- The accumulators we will use are based on Merkle trees:
  - The summary is the root (or collection of roots).
  - The witnesses are authentication paths.
- We’ll need accumulators for note commitments, and nullifiers.
  - Zcash already uses a Merkle tree accumulator for note commitments.
  - We need to add one for nullifiers, because we need to verify that notes have not been spent before within a circuit.
- In both cases, knowing which witnesses a client is interested in can leak private information.

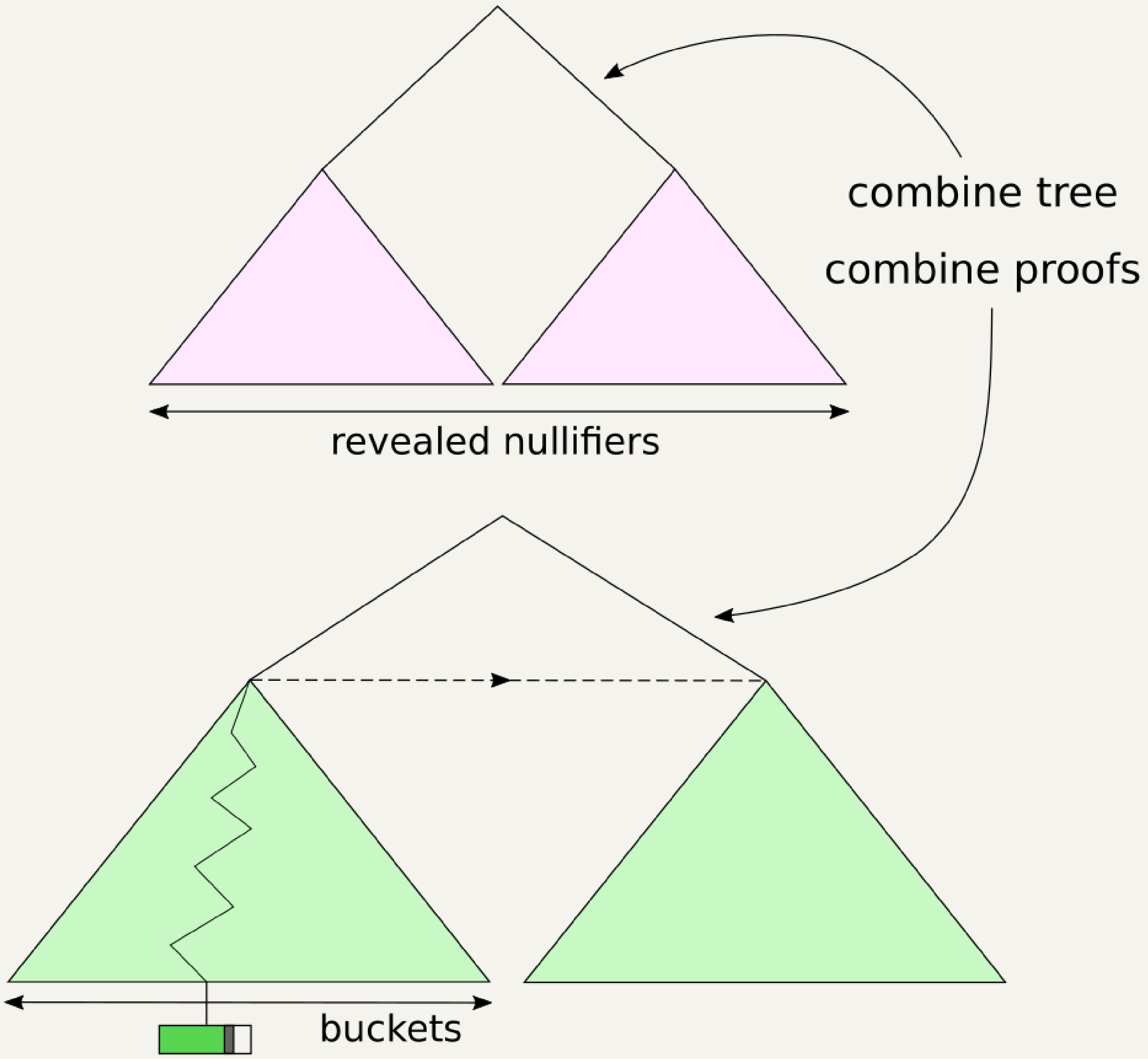
Nullifier accumulator

- Detecting concurrent double spends seems to require merging information across shards. But does it really require that?
- Straw proposal:
  - Partition the nullifier space, assigning each partition to a shard.
  - To spend several notes, they must be in the same shard. Each transaction is sent to the right shard for the notes it is spending.
  - If we don’t have enough value in a single shard, we have to move it in a separate transaction.
  - The nullifier accumulator can be updated at the same time as aggregating transactions, without inter-shard communication. This is partly sequential, but the actual proofs can be made in parallel.
- This solution does not require token holders to maintain witnesses for the nullifier accumulator.
- There was another approach I decided to omit from the talk, with transactions and nullifier partitions sharded separately. Ask me about it in the scaling workshop.
- There are privacy concerns about transactions being linked (e.g. if a wallet were to preemptively move all its notes to the same shard). So privacy depends to some extent on wallet behaviour.

Nullifier accumulator

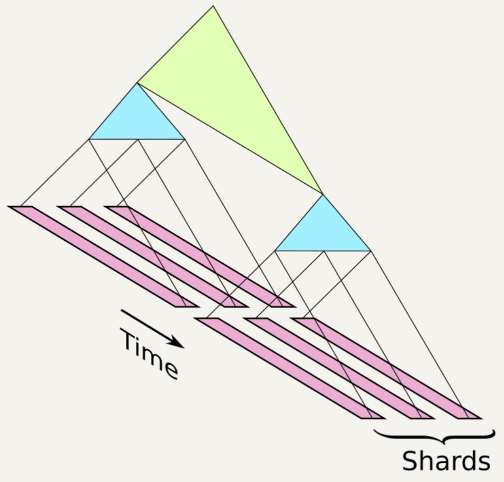
- A leaf proof represents the operation “add a nullifier provided that it did not already exist in the set”.
  - This has to be combined with a proof that the nullifier is correct for some Spend, but that’s already part of the Sapling Spend circuit; here we’re concentrating on the new part.
- Existing constructions for use outside circuits tend to use optimizations that don’t work efficiently in a circuit.
- We can use a sparse bucketed Merkle tree.
- Nullifiers already have a small probability of clashing by accident due to hash collisions. We can limit the size of buckets provided that the probability of any bucket becoming full is negligible.

Nullifier accumulator (simplified)



Note commitment accumulator

- The note commitment accumulator has to support addition, and membership proofs.
  - It is sequential-access, i.e. we only add commitments “at the end”.
  - This allows some nice scalability optimizations: we pay a small cost to increase the depth of the tree, then we can add commitments in batches (as a Merkle subtree per block, say) rather than individually.
  - Witnesses for held notes then only need to be updated per block – not too much of a problem.
  - Note holders must keep their witnesses secret and update them secretly, otherwise they leak when the note was created.
  - But, only the per-block portion of the witness (the green part here) changes.
  - This part could possibly be represented as a Merkle Mountain Range to further reduce overhead.



Protocol bottlenecks revisited

- All full validators must update the commitment tree. Per shard
  - The commitment tree is global, but updates are to a particular shard's subtree.
- All full validators must update the nullifier set. Per shard
  - Nullifier update proofs are concretely quite expensive. Unclear how much of a problem this is.
- All full validators must validate each transaction. Per shard
- All transactions must be broadcast to every full validator. Per shard
- All transactions must be broadcast to every potential token-receiver.
- All token-receivers must check each output in case it is sent to them.
- Miners must validate ~~transactions~~ summaries for inclusion in blocks. O(1) per shard
- Blocks must be broadcast. Per shard
  - A block includes its transactions, each of which must have been broadcast at some point. Per shard
- Full validators that have been off-line must catch up to the current head. Per shard
- Fast validators need only verify *one* SNARK proof, and update note witnesses per block.

Protocol bottlenecks revisited

- Current Zcash includes an anonymous communication protocol that uses the P2P network to broadcast note ciphertexts, encrypted with key-private Diffie-Hellman.
- Advantages: simplicity; ideal “on-chain” privacy; clients can recover incoming payments after forgetting all but their private key seed.
- Flood-broadcasting all ciphertexts doesn’t scale.
- Requesting ciphertexts (if you don’t receive everything) leaks which transactions you’re interested in.
- Clients have to trial-decrypt all ciphertexts.
- No transport privacy: transactions are submitted in the clear.
- Transport privacy is useful for several reasons:
  - Even when transactions are fully shielded, some metadata is leaked (e.g. number of Spends and Outputs).
  - If I’m the recipient of a payment (or have the viewing key) and *also* a passive global adversary, I can see where that transaction entered the network.
  - Defence in depth.

Scalable anonymous communication

- There have been proposals for horizontally scalable mix-nets. For example, Atom: Horizontally Scaling Strong Anonymity (2017) .
- Basic idea: verifiable mix-net, in which each node is an “anytrust group”.
- When submitting a message, use a non-malleable NIZK to prove you knew the plaintext. (This prevents replay attacks.)
- Use ElGamal reencryption to shuffle ciphertext batches; prove within each group that they are correctly shuffled.
- Communication cost is O(M/N) → horizontally scalable.
- This is an active area of research; there might be better proposals. Loopix is also interesting.
- Don’t miss David Stainton’s workshop on mix-nets tomorrow (June 23).
  - Thanks Antonie, for moving the scaling workshop to avoid a clash.

Payment protocol

- How does a payer verify *whether or not* its transaction got into the chain?
  - In current Zcash or pre-segwit Bitcoin, it effectively can't because transactions are malleable. But we can fix that (it's simpler for shielded-only).
- Better: How does a payer verify that its note was spent in a given transaction?
- Modern mix-net designs –Mixminion and later– support "Single Use Reply Blocks" (SURBs). Each SURB allows a message-recipient to reply to a sender once without breaking anonymity.
- The destination shard can use this to reply with a *receipt* that the payer can use to determine whether the transaction got into the chain.
  - Unresolved: The shard could include your transaction without sending back a valid receipt.
- The payee also gets a message (could be either from the payer or the shard) containing a note.
  - Payer has incentive to retry sending to the payee until they confirm.
  - If payee doesn't confirm but keeps the money, then the receipt can be used in disputes.
  - We also want to support "payer present" applications where the transaction is transmitted directly to the recipient.
- Unresolved: How do I trial-decrypt with a viewing key without downloading all transactions?

Bonus slide: Concrete efficiency

- We need curves suitable for recursive SNARKs.
- There are two known ways of constructing these:
  - MNT-4/6 cycles;
  - Cocks-Pinch or Dupont-Enge-Morain.
- Only the MNT-4/6 cycles work for unbounded recursion. Coda uses these.
- They require a $\mathbb{G}_1$ size of ~1143 bits (3 times larger than BLS12-381).
  - The reason for this is that the embedding degree is 3 times smaller (4 in the worst case rather than 12), so $\mathbb{G}_1$ must be 3 times larger to get the same target field size and security.
- This is likely to result in a 10-20 times efficiency loss.
- Fortunately, there is a ~10 times improvement in Spend circuit size available by using circuit-efficient hashes (e.g. Rescue).
  - This depends on these hashes getting sufficient cryptanalysis by the time we need to rely on them. One way to encourage that would be to sponsor a competition.
- Who's up for a BoF / freeform workshop on Day 3 about pairing-friendly curve construction?

Scalable Privacy

Daira Hopwood
<daira@electriccoin.co>
@feministPLT
@daira on chat.zcashcommunity.com
Slides at https://github.com/daira/zcon