A Note on Notes: Towards Scalable Anonymous Payments via Evolving Nullifiers and Oblivious Synchronization

Sean Bowe[1] and Ian Miers[2]

[1] Zcash ewillbefull@gmail.com
[2] University of Maryland imiers@umd.edu

Abstract. Anonymous payment protocols based on Zerocash (IEEE S&P 2014) have seen widespread deployment in decentralized cryptocurrencies, as have derivative protocols for private smart contracts. Despite their strong privacy properties, these protocols have a fundamental scaling limitation in that they require every consensus participant to maintain a perpetually growing set of *nullifiers*— unlinkable revocation tokens used to detect double-spending—which must be stored, queried and updated by all validating nodes. This set grows linearly in the number of historic transactions and cannot be discarded without the undesirable effect of destroying unspent funds.

In this short note, we introduce a new technique that enables continual, permanent pruning of nullifiers by validators, without imposing significant computation, bandwidth or latency overhead for users, and without compromising privacy. Our main contribution is a general model we call *oblivious synchronization* whereby users ask untrusted remote services (which ingest and process the public ledger) to create succinct proofs that coins are unspent and otherwise valid. Crucially, these services are fully oblivious to their clients' transaction details and cannot link their clients to any transactions that ultimately appear on the public ledger. Moreover, these services only keep ephemeral state per client and users can freely switch between services without incurring redundant computational effort.

1 Introduction

Anonymous payment protocols based on Zerocash [6] have seen widespread deployment in cryptocurrencies and distributed ledger protocols including directly in Zcash [11] and a number of follow on projects including Tornado.cash[1] [17], Railgun [19], Ironfish [12], Nomada [4], PrivacyPools [18] and others. Unlike many non-private payment schemes where transaction details are exposed publicly, these protocols leverage zero-knowledge proofs (specifically zk-SNARKs)

[1] While Tornado Cash had experimental support for full-fledged payments [20], its core design is a hybrid. It combines the trustless zk-proof mixer concept from Zerocash's predecessor, Zerocoin [15], with the SNARK and Merkle tree techniques from Zerocash.

so that users can *prove* their transactions satisfy certain correctness criteria without revealing details such as the identity of the sender, the recipient, or the amount. These techniques have been extended to privacy preserving smart contract systems both deployed and in active commercial development.

The general approach of Zerocash and related protocols is to first encapsulate spendable value into units of account called *notes*, each of which consists of a numerical amount of funds (similar to a bank account balance), its owner's identity, and possibly some metadata. Payments are made via transactions that consume existing notes and create new ones. For example, if Alice pays Bob $2, a transaction might consume a note owned by Alice containing $10 and create a new note owned by Bob for $2 and another owned by Alice for her remaining $8 in "change".

Unlike standard blockchains, where transaction data is public and checked by validators, in Zerocash-derived systems all such details are private. Instead, transactions include cryptographic commitments to new notes, a zero-knowledge proof that the transaction is correct, and, crucially, a *nullifier* to prevent double-spending. To verify the transaction, validators verify the proof and check that the nullifier is unique, which prevents a client from spending the same money twice.

Valid transactions have their note commitments inserted into an accumulator. Canonically in Zerocash, the accumulator is a Merkle tree, though we will refer to it abstractly as other approaches are possible. The combination of the nullifier and accumulator means that, in Zerocash-derived schemes, the blockchain does not reveal when or how money moves in the system. In particular, a zero-knowledge proof alone is not sufficient. The proof can check operations that manipulate the state of the system, but on its own, it cannot prevent replay or forking of old states.

In more detail, the zero-knowedge proof can be viewed as a program that checks that:

Existence Notes being spent are well-formed and commitments to them exist somewhere within the global accumulator. The state of this accumulator (e.g., the Merkle tree root) is a public input to the proof called the *anchor*, but the actual witness for membership in this accumulator (e.g., Merkle path) and the notes being spent are private information that is not revealed by the proof.

Authorization The client knows the secret keys corresponding to the identity in the note or equivalently, that (ephemeral) key material located in the transaction can be used to establish this authority.

Balance Integrity The total monetary value of the notes being created (and any residual value used as a fee) must not exceed the total value of the notes being spent, to prevent counterfeiting.

Nullifier Correctness The nullifiers that appear in the transaction must be computed correctly (and deterministically) based on the notes being spent. Ideally, a nullifier is an output of a pseudo-random function applied to the

note and keyed on a secret known to the spender, though additional cryptographic properties may be desired.

Abstractly, we can view transaction validation as two programs. One run by the transaction author that performs local checks. And a second program that performs global checks, run by the validators. To ensure the checks in the first program are correct, clients produce a zero-knowledge proof. To ensure the checks in the second program are correct, each validator executes that program. Accumulators and nullifiers are a crucial communication boundary between those two programs, through which the validators learn that the note being spent exists and was not already spent, but do not learn the note's (and hence the sender's) identity.

We note that this same approach, and the scaling limits we describe below, also apply to privacy-preserving smart contract systems. However, in such systems, the specific "balance integrity" check is replaced by a more general execution kernel that verifies both arbitrary application logic and, crucially, facilitates interaction between mutually distrusting applications. In particular, Zexe [8], which introduced techniques for a full-fledged privacy-preserving smart contract system with cross-contract calls. Derivatives of this approach are in commercial deployment or under active commercial development by projects including atleast Aleo [3], Aztec [5], and Miden [14].

1.1 Scaling Anonymous Payments

Zerocash-derived protocols depend, generically, on some consensus mechanism. In cryptocurrency terms this may involve proof of work, proof of authority, or proof of stake. In distributed systems terms, this is state-machine replication.

The validation logic for this state machine—verifying proofs and checking for double-spent nullifiers—is relatively lightweight. This makes Zerocash schemes far more scalable than smart contract systems like Ethereum, where validators must re-execute every smart contract. And in Zexe, this holds even if we extend the functionality to smart contracts, since validators need only verify a proof of contract execution.

However, compared to simple, non-private payment systems like Bitcoin (which only require checking a signature and TXO status), Zerocash introduces two distinct scaling costs. The first is proof verification, which is far more expensive than checking an ECDSA signature. This cost, however, can be amortized away as individual transaction proofs can be *aggregated* into a single batch proof (e.g., via recursive proofs). In the limit, the validation cost for an entire block could be reduced to verifying just one proof.

The second (and more significant) cost is that validators must still track nullifiers. The perpetually growing nullifier set is a major scaling bottleneck for Zerocash-derived protocols as we scale beyond existing cryptocurrency applications since it grows linearly with the number of transactions that have ever occurred. As an example, assume a transaction consumes two notes, revealing two 32-byte nullifiers. At 100 transactions per second, the nullifier set would

grow by half a GB each day. At the transaction volumes for commercial payment systems like Visa, it would be considerably larger.[2]

Nullifier data must be stored online and readily accessible to allow double-spend checks. In contrast, in most other payment systems, transaction history can be either offloaded from the hot path or simply stored in an archival node. As such, it represents a unique scaling limitation for anonymous payment systems if they are pushed well beyond current cryptocurrency scale or used in a centralized e-cash setting to gain auditability.

The nullifier scaling challenge has been known since Zerocash's publication. And indeed, is not unique to Zerocash, but applies to both Zerocash-derived private payment protocols and many privacy preserving smart contract systems derived in follow up work [8; 13]. Various approaches have been discussed for pruning the nullifier list. One simple approach is to expire both notes and nullifiers once they are too old.[3] Unfortunately, users must move their funds into a new note, once per window, or lose their funds. At very large scale, this process might need to be done weekly or even daily depending on the minimum resources required for a validator.

A more promising approach is modifying the zero-knowledge proof in a transaction to check if the note is already spent. In these proposals an accumulator is provided for nullifiers, and the proof computes the nullifier and checks that the nullifier is not already in the accumulator. Since blockchains are organized into blocks, there is a clear boundary for past transactions. It is still necessary to reveal the nullifier, as we must prevent *concurrent* double spends in the same block, for which validators must still check for duplicates. And in practice, we may make unspent proofs relative to an older block to increase stability.

Various proposals for handling the nullifier problem have been discussed, but as far as we know most have not been formally written up, proposed or deployed[4]. The most promising techniques we're aware of essentially center around the idea of periodically freezing the active nullifier set for an "epoch" of transactions in the ledger, collecting it into an accumulator, and requiring users provide non-membership proofs for prior epochs when spending their notes. One such proposal can be found here[1]. Unfortunately, this approach runs into a frequently undiscussed challenge in scaling zerocash-like protocols: accumulator updates.

---

[2] Visa completed 322 billion transactions last year, giving about 10,000 transactions per second `https://corporate.visa.com/content/dam/VCOM/corporate/visa-perspectives/documents/about-visa-factsheet.pdf` and claims capacity for 65,000 TPS.

[3] Since a nullifier spending a note appears after the note's creation, we can safely remove a nullifier if we also remove all notes created before its appearance.

[4] Unfortunately there've been nearly a decade of such discussions by various folks in forums, DMs, etc. Including abandoned proposals by the authors. Any list of such cites would be incomplete and in fact almost certainly miss the origin of the ideas by years.

1.2 The Tyranny of the Accumulator Update Equations

To prove membership or non-membership in an accumulator requires a witness w (e.g., siblings on the path from leaf to root in a merkle tree). The challenge in Zerocash-like payment systems is that this witness must be updated as notes or nullifiers are added to the accumulator. As a result, naively, each client must process every block of transactions in order to update their witnesses. At any reasonable scale, this is cost prohibitive.

For membership proofs, once a client has an initial witness to their note's existence, updates to membership proofs can be batched, saving the client a constant factor. However, for non-membership proofs, e.g., to prove that a nullifier is not already spent, we run into an additional problem: clients cannot compute the initial non-membership proof without access to the full nullifier history. There are a variety of tricks to reduce this problem, but known lower bounds for accumulators that make it difficult to eliminate [10].

A natural approach is to outsource this processing to a third party. However, this raises privacy concerns, since the process of computing an accumulator witness (for membership of a note or non-membership of a nullifier) leaks exactly the element the proof is for. In particular, a client outsourcing such operations for multiple notes/nullifiers would link those notes together and reveal they are owned by the same user. This is a fundamental privacy violation. And, especially since these outsourcing nodes, in many deployment scenarios, would be volunteers or part of the cryptocurrency network itself, it represents a substantial opportunity for surveillance, e.g, via a Sybil attack on the network.

1.3 A New Approach: Evolving Nullifiers

Scaling anonymous payments hinges on solving the nullifier problem, but existing proposals face a critical flaw. They create an intractable dilemma by forcing a choice between two non-starters: (a) requiring clients to download and store the entire, ever-growing set of spent nullifiers just to prove their own is unspent, or (b) a complete loss of privacy when outsourcing the generation of that proof. Our approach resolves this conflict not by trying to fix the accumulator, but by making the nullifier itself evolve over time, allowing old versions to be safely revealed.

As a concrete example, assume the nullifier for preventing the double-spend of a note is modified from $\eta \leftarrow PRF(k_{\text{note}}, \rho)$ to $\eta_e \leftarrow PRF(k_{\text{note}}, \rho||e)$ where k_{note} is a key bound to the note, ρ is a nonce, and e is a monotonic counter known as an *epoch identifier*. Without loss of generality, we will assume it is incremented per block, though in practice we expect it to change less frequently. e is revealed publicly in spends and validators enforce the epoch is current in addition to the existing double-spend check.

Evolving nullifiers effectively divide the ledger into epochs. For any attempted double-spend of a note twice (or more) in a single epoch, the existing double-spend check by the validators suffices, since the same nullifier is deterministically

generated for all spends in an epoch and validators maintain a database of that epoch's spent nullifiers.

To prevent double-spending across epochs, we cannot rely on such a check precisely because the nullifier changes. Instead, we must ensure that every previous version of the nullifier is also unspent. Revealing previous versions of the nullifier would allow validators to check this in the clear, but then we are back to our core scaling challenge: validators store all spent nullifiers for every epoch/block, and now they need to perform more checks against it. Instead, we outsource this check to an untrusted third party *oblivious syncing service* who produces an *incremental unspent proof* that clients incorporate into their transactions. Because nullifiers evolve and different versions are unlinkable, these old nullifiers can safely be outsourced while the client maintains the latest nullifier(s).

Observe two key points about this incremental unspent proof. First, as the name implies, it can be computed incrementally. Given a proof that previous nullifier versions up to epoch $i-2$ were unspent and a proof that the nullifier version for $\eta_{e_{i-1}}$ is well-formed, we can create a new proof (via recursive proof composition) that the nullifier for epoch $i-1$ was also unspent. Second, this proof requires no sensitive information from the client; its components are only what the client would have revealed *if* they had spent in epoch e_{i-1}. So each incremental update to this proof can be computed by a completely untrusted party.

Readers will note, however, that some care must be taken with regards to the timing of user actions. The primary concern is that wallets hold multiple notes, and thus may update their unspent proofs at the same (or correlated) time, such as when first turned on after a period of inactivity, or charging on WiFi at home. Transactions spending these notes to different parties could still be linked together by their use of incremental unspent proofs that were computed at correlated times. This is not a small problem; in fact, it would be fatal to the entire approach. To preserve the privacy of Zerocash-derived schemes, it is essential that the syncing service be incapable of distinguishing between transactions they assisted with and ones they did not. This problem can be generally addressed by re-randomizing the proof, as many recursive proof systems permit, such as those involving accumulation schemes[9].

1.4 Beyond Simple Outsourcing: Batched Proofs and a Data Availability Layer

Our evolving nullifier approach raises two interesting points. First, the possibility of batching the incremental unspent proofs. Here, the synchronization service can compute one proof for a batch of disparate transactions from different clients. Various efficient batched non-membership proofs exist [7; 16] and there is the possibility of co-designing a specific proof system and accumulator to improve performance.

The second possibility is to build a form of data-availability (DA) layer [2] for Zerocash-like protocols, allowing us to move beyond ad-hoc oblivious syncing services. In particular, a DA layer would allow the network to still store old

nullifiers, avoiding a dependency on the goodwill or business interests of others, without paying the high cost of forcing the main chain's consensus nodes to store it forever. Indeed, this is the general idea of a DA layer, which separates the job of ordering transactions from the job of guaranteeing that the transaction data is available for anyone who needs to check it.

This would leave us with a transaction layer, which executes transactions by validating proofs (including the incremental unspent proofs of previous epochs) and checks double spends for that epoch, and a separate data-availability layer which stores the nullifier database for past epochs and operates oblivious syncing services to help users make transactions.

These systems could readily be set up as a Proof-of-Stake (PoS) network independent of the consensus mechanism for the underlying transaction layer (which, e.g., in Zcash is Proof-of-Work). As the unspent nullifier list is simply data, there is nothing special required of the DA layer.

The need for such a data availability layer is a direct function of epoch length. With long epochs (e.g., years), a cryptocurrency ecosystem can perhaps organically support this need through exchanges, wallet providers, and hobbyists running syncing services and storing nullifiers for past epochs. However, to achieve a scale approaching that of traditional payment providers, epochs must become much shorter—perhaps even as short as a single block. As the system grows and scales, one could easily imagine a point where the historical nullifier database grows so large that it can only be stored by a few highly centralized providers, like major exchanges or wallet providers. This could introduce a critical point of failure, where the sudden shutdown of these services could render the funds of all their dependent users unspendable, potentially permanently.

References

1. `https://forum.aztec.network/t/global-state-epochs/2704`
2. Al-Bassam, M.: Lazyledger: A distributed data availability ledger with client-side smart contracts. CoRR **abs/1905.09274** (2019), `http://arxiv.org/abs/1905.09274`
3. Aleo Team: Aleo: The Layer 1 for Private Applications. `https://aleo.org/`
4. Anoma Foundation: Namada: An L1 for Interchain Asset-Agnostic Privacy. `https://namada.net/`
5. Aztec Labs: Aztec: The Privacy-First L2 on Ethereum. `https://aztec.network/`
6. Ben Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: Decentralized anonymous payments from bitcoin. In: 2014 IEEE Symposium on Security and Privacy. pp. 459–474 (2014). https://doi.org/10.1109/SP.2014.36
7. Boneh, D., Bünz, B., Fisch, B.: Batching techniques for accumulators with applications to iops and stateless blockchains. In: Advances in Cryptology – CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part I. p. 561–586. Springer-Verlag, Berlin, Heidelberg (2019)
8. Bowe, S., Chiesa, A., Green, M., Miers, I., Mishra, P., Wu, H.: ZEXE: Enabling decentralized private computation. In: 2020 IEEE Symposium on Security and

Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020. pp. 947–964. IEEE (2020). https://doi.org/10.1109/SP40000.2020.00050, https://eprint.iacr.org/2018/962
9. Bünz, B., Chiesa, A., Mishra, P., Spooner, N.: Proof-carrying data from accumulation schemes. Cryptology ePrint Archive, Paper 2020/499 (2020), https://eprint.iacr.org/2020/499
10. Christ, M., Bonneau, J.: Limits on revocable proof systems, with applications to stateless blockchains. Cryptology ePrint Archive, Paper 2022/1478 (2022), https://eprint.iacr.org/2022/1478
11. Company, E.C.: Zcash: Privacy-protecting digital currency, https://z.cash/
12. Iron Fish Team: Iron Fish: The Universal Privacy Layer for Web3. https://ironfish.network/
13. Kosba, A.E., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. In: IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016. pp. 839–858 (2016). https://doi.org/10.1109/SP.2016.55, https://doi.org/10.1109/SP.2016.55
14. Miden Team: Miden is the Edge Blockchain. https://miden.xyz/
15. Miers, I., Garman, C., Green, M., Rubin, A.D.: Zerocoin: Anonymous distributed e-cash from bitcoin. In: Proceedings of the 2013 IEEE Symposium on Security and Privacy. pp. 397–411. SP '13, IEEE Computer Society, Washington, DC, USA (2013), http://dx.doi.org/10.1109/SP.2013.34
16. Ozdemir, A., Wahby, R., Whitehat, B., Boneh, D.: Scaling verifiable computation using efficient set accumulators. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 2075–2092. USENIX Association (Aug 2020), https://www.usenix.org/conference/usenixsecurity20/presentation/ozdemir
17. Pertsev, A., Semenov, R., Storm, R.: Tornado cash privacy solution version 1.4, https://berkeley-defi.github.io/assets/material/Tornado%20Cash%20Whitepaper.pdf
18. Privacy Pools Team: Privacy Pools. https://privacypools.com/
19. RAILGUN Contributors: RAILGUN: Privacy System. https://railgun.org
20. Tornado.cash Team: Tornado.cash Nova. https://github.com/tornadocash/tornado-nova