Zether: Towards Privacy in a Smart Contract World

Benedikt Bünz[1], Shashank Agrawal[2], Mahdi Zamani[2], and Dan Boneh[1]

[1] Stanford University, CA USA
[2] Visa Research, Palo Alto, CA USA
{benedikt,dabo}@cs.stanford.edu, {shaagraw,mzamani}@visa.com

Abstract. Smart contract platforms such as Ethereum and Libra provide ways to seamlessly remove trust and add transparency to various distributed applications. Yet, these platforms lack mechanisms to guarantee user privacy, even at the level of simple payments, which are essential for most smart contracts.

In this paper, we propose *Zether*, a trustless mechanism for privacy-preserving payments in smart contract platforms. We take an account-based approach similar to Ethereum and Libra for efficiency and usability. Zether is implemented as a smart contract that keeps account balances encrypted and exposes methods to deposit, transfer, and withdraw funds to/from accounts through cryptographic proofs at only a small cost.

We address several technical challenges to protect Zether against replay attacks and front-running situations and develop a mechanism to enable interoperability with arbitrary smart contracts, making applications like auctions, payment channels, and voting privacy-preserving. To make Zether efficient, we propose Σ-Bullets, a zero-knowledge proof system that is optimized for Σ-protocols. We implement Zether as an Ethereum smart contract and show its practicality by measuring the amount of gas used by the Zether contract. A Zether confidential transaction costs about 0.014 ETH or approximately $1.51 (as of early 2019), which can be drastically reduced with minor changes to Ethereum that we describe in the paper. The full version of the paper is available online [BAZB19].

1 Introduction

Smart contracts are computer programs that can directly control digital assets [Sza96], and hence automate the execution of operations that involve digital payments such as digital auctions, lotteries, and crowd-sales. Following the rise of cryptocurrencies, blockchain-based smart contract platforms such as Ethereum [etha] and Libra [lib19] enable execution of smart contracts in a decentralized, transparent fashion, removing/reducing the liabilities of trusted intermediaries.

A smart contract is typically written in a powerful programming language, such as Solidity [Sol] or Move [mov19], and is executed over a replicated state that is visible to the public. While this allows anyone to automatically verify the correct execution of the contract, it can expose sensitive user data to untrusted entities. One may choose to simply encrypt *all* the state data to avoid such exposures.

Unfortunately, this makes the verification process significantly expensive, leading to massively-high execution fees.

In contrast, depending on the application, one may choose to encrypt *only* the information pertaining to the transfer of assets (i.e., payments) happening as part of the contract execution. In fact, in many scenarios, especially those involving competitive risks such as stock trading and auctions, payments information (i.e., amounts and identities of the senders/recipients), are the main source of privacy concerns. Unfortunately, existing techniques for confidential and anonymous payments, such as Monero [Noe15] and Zcash [zcaa], do not easily and efficiently extend to smart contract payments, and popular smart contract platforms such as Ethereum do not provide any privacy mechanism. Furthermore, existing privacy-preserving smart contract mechanisms, such as Hawk [KMS+16] and Ekiden [CZK+18], are not completely trustless (see Section 3 for details).

Most existing payment confidentiality mechanisms (e.g., [Max15, Noe15, MGGR13, BCG+14, zcaa]) are in the *unspent-transaction-output (UTXO)* model popularized by Bitcoin. In this model, the inputs to a new transaction are the unspent outputs of previous transactions. UTXOs are not well-suited for applications that need to maintain some state [But16], so smart-contract platforms like Ethereum and Libra operate in the account-based model. Another drawback of existing UTXO-based mechanisms is that they require major changes to the design of the underlying cryptocurrency (typically Bitcoin), and thus have spun off into separate cryptocurrencies. An immediate benefit of smart contract platforms like Ethereum is that allow deploying new applications without much changes to the underlying blockchain protocol.

Our Contribution. We propose *Zether*, a fully-decentralized, privacy-preserving payment mechanism in the account-based model. Zether requires no changes to the design of the underlying smart contract platform (e.g., Ethereum). As such, the techniques used in Zether can apply to other account-based cryptocurrencies, completely independent of their blockchain/consensus mechanisms.

Our contributions can be summarized as follows:

- *Confidentiality.* Transactions on Zether are confidential by design. Account balances are kept encrypted at all times and users provide cryptographic proofs to spend their money
- *Anonymity.* Zether allows anonymous transfers, i.e., can hide the sender and the receiver of a transaction among a group of users chosen by the sender. Our protocol neither requires any trusted setup nor any changes to the underlying smart contract platform.
- *Zero-Knowledge Proofs.* To make Zether efficient, we propose a new zero-knowledge (ZK) proof mechanism, called **Σ-Bullets**, which enhances the interoperability of Σ-protocols [Dåm] and Bulletproofs [BBB+18] to perform range proofs with ElGamal encryptions efficiently.
- *Implementation.* We implement Zether as an Ethereum smart contract and measure the gas amount required for executing it. We show that Zether is

practical today and with already-planned enhancements to Ethereum will become even more efficient.

– *Interoperability.* Zether allows locking an account to a smart contract, making it easy to "add" privacy to existing applications. We show how Zether can be used to perform sealed-bid auction, confidential payment channel, confidential stake-voting, and private proof-of-stake.

2 Overview of Zether

The Zether design consists of a smart contract, called the *Zether contract (ZSC)* that manages Zether tokens, denoted by ZTH. The contract maintains an encrypted account information, referred to as a *Zether account*, for any user who wishes to transact privately using ZTH over the underlying smart contract platform.

To make payment transactions confidential, several proposals (e.g., [Max15, Noe15, Poe16]) use homomorphic commitments, such as Pedersen commitments [Ped92]. Though such commitments are simple and efficient, the opening of these commitments must be transferred to the receiver, say Bob, so that he can spend the money later. This randomness could be stored on-chain in some encrypted manner or sent directly to Bob through a separate channel. In the UTXO model, if Bob is unable to recover the randomness (an incorrect value was encrypted/sent, nothing sent at all, etc.), then it cannot spend the UTXO later. However, other UTXOs controlled by Bob are not affected at all and could still be spent. On the other hand, with an account-based model, since all the incoming transfers go into the same account, failure to recover the randomness for even a single transfer could render the whole account unusable. One could require senders to encrypt the randomness under receivers' public key, and prove that the commitment indeed uses the randomness encrypted.

Zether uses ElGamal encryption with messages in the exponent [CGS97] to achieve homomorphism and create efficient ZK-proofs of correct encryption. Zether accounts are identified with ElGamal public keys which are stored in the contract's internal state. To fund an account with public key y with b ZTH, the user sends b ETH to ZSC which generates an ElGamal encryption of b with randomness 0 and "adds" it to the encrypted balance associated with y.[3] The user can convert ZTH back to ETH by revealing the current balance $b^\star$ and providing a ZK-proof that y's ciphertext indeed encrypts $b^\star$.

In order to transfer some b amount of ZTH to a public key y' without revealing b itself, one can encrypt b under both y and y'. A ZK-proof is provided to show that the two ciphertexts are well-formed and the remaining balance associated with y is positive. Zether relies on a new ZK-proof system, called Σ-Bullets, to efficiently prove correctness statements over the encrypted transfer balance and the new sender balance.

While the above design is simple and efficient, it introduces multiple challenges which we briefly discuss in the following.

[3] If y has no record on ZSC yet, then a new record is created and initialized with the aforementioned ciphertext.

Front-Running Problem. In Zether, ZK-proofs are generated with respect to a certain state of the contract. For example, the ZK-proof in a transfer transaction needs to show that the remaining balance is positive. A user, Alice, generates this proof with respect to her current account balance, stored in an encrypted form on the contract. Unfortunately, if another user, Bob, transfers some ZTH to Alice, and Bob's transaction gets processed first, then Alice's transaction will be rejected because the proof will not be valid anymore. This can happen even if Bob is totally benign and yet Alice loses the fees she paid to process her transaction. We refer to this situation as the *front-running* problem. Burn transactions have a similar problem, too: a proof that a ciphertext encrypts a certain value becomes invalid if the ciphertext changes.

To solve this problem, one could introduce a new type of transaction that just locks an account to keep away incoming transfers. Alice could wait until this transaction gets into the blockchain before initiating an outgoing transfer (or doing a burn). While this seems to fix the problem (at the cost of making transfer, the primary transaction, a two-step process), it creates new problems for users like Bob who want to send ZTH to Alice. Alice's account may not be locked when Bob publishes a transfer transaction tx, but it could get locked before tx gets in, resulting in tx being rejected.

Pending Transfers. To address the front-running problem, we keep all the incoming transfers in a *pending* state. These transfers are rolled over into the accounts from time to time so that the incoming funds could be spent. This rollover cannot happen at arbitrary times, otherwise the proofs would get invalidated again. To handle this, we divide time into *epochs* each consisting of k consecutive transaction blocks. The choice of k depends on two factors: (1) The gap between the latest state of blockchain and any user's view, and (2) the time it takes to get a transaction into the blockchain. At the end of every epoch, pending transfers are rolled over into the corresponding accounts.

Unfortunately, a smart contract does not do anything unless a transaction is sent to it. One may rollover the pending transfers for *all* accounts on the receipt of the first message in an epoch. This, however, places an unreasonably large burden on the sender of that message: it will have to pay for the cost of rolling over the accounts that it does *not* own, which could be too many. Furthermore, users would have no way to know if their transaction would be the first in an epoch, so they cannot estimate the right amount of gas to supply. To avoid this, Zether rollovers an account in an epoch when the first message from *this* account is received; so, one message rolls over only one account. Note that there could be accounts that do not get rolled over for several consecutive epochs because no transaction is initiated from them.

Replay Protection. Ethereum provides replay protection of its own by associating nonces with every account, which need to be signed into every transaction. Unfortunately, this level of protection is not enough for Zether because: (1) Zether accounts have their own public keys which are not associated with Ethereum addresses, and (2) Zether transactions contain *non-interactive* ZK-proofs. A malicious actor can steal these proofs and reuse them in new transactions. If the state of the account has not changed, then the new transactions will also be processed successfully, leading to loss of funds.

To protect against such issues, we associate a nonce with every Zether account. The nonces are incremented as transactions are processed. A new transaction from an account must sign the latest value of the nonce associated with the account along with the transaction data, which includes any ZK-proof. This approach binds all components of a transaction together and ensures freshness. ZK-proofs cannot be imported into malicious transactions and valid transactions cannot be replayed.

Anonymous Transfers. To allow anonymous transactions in Zether, we require more complex ZK-proofs, a new replay and double-spend protection mechanism, and a new mechanism to lock accounts to smart contracts. An anonymous transaction allows a user, Alice, to send some b ZTH to another user, Bob, while hiding both her and Bob's identity among a larger group of n users. Alice generates n ciphertexts C_1,...,C_n, one for each member of the group, respectively and provides a ZK-proof, π, showing that all the ciphertexts encrypt 0 ZTH except two of them which encrypt b ZTH but with difference signs, i.e., b and $-b$. Also, the proof shows that the remaining balance of the account with positive amount is non-negative.

A major challenge in providing anonymity is the size of the new ZK-proof, π, which increases linearly with the size of the anonymity set, n. Zether provides several optimizations to reduce the size of π and its verification overhead. Namely, each ZK-proof contains only two range proofs that are computed using the one-out-of-many proofs of Groth and Kohlweiss [GK15]. These proofs can be used to give a secondary encryption to one out of n ciphertexts without revealing which original ciphertext was re-encrypted. One-out-of-many proofs can be used to build ring-signatures. Alice uses this proof to create secondary encryptions of b and $-b$, respectively along with a secondary encryption of Alice's balance b^*. Alice then shows the relationship between b and $-b$ and that b and $b^* - b$ are non-negative using a range proof.

Σ-Bullets. Zether ensures that encrypted transactions are correct by using ZK-proofs that certify correctness without revealing any additional information. We design a custom proof system called Σ-Bullets that is well-suited for Zether. Σ-Bullets integrate Bulletproofs [BBB$^+$18] with Σ-protocols to enable efficient proofs on algebraically-encoded values such as $\exists x : g^x = y \wedge h^x = u \in \mathbb{G}$. Bulletproofs on the other hand is a circuit proof system that is well suited for range proofs and other more complicated arithmetic statements. Bulletproofs does enable proofs on Pedersen committed values if all values use the same commitment key. With Σ-Bullets, we can efficiently prove that a set of ElGamal encrypted values are in some range. Further, we combine one-out-of-many proofs [GK15], also known as ring signatures, with range proofs to allow anonymous transfers. The one-out-of-many proof is a Σ-protocol that hides which account is being used. Bulletproof is then used to show that the account has sufficient funds for the transfer.

Σ-Bullets inherits from Bulletproofs the trapdoor-free setup and the short, logarithmic sized, proof lengths. The ability to prove statements on encrypted values further significantly reduces the prover and verifier time compared to a naive implementation using Bulletproofs. We describe Σ-Bullets in detail in Appendix E.

3 Related Work

Confidential transactions for Bitcoin were first proposed by Maxwell [Max15] who used Pedersen commitments [Ped92] and OR-proofs to hide transaction amounts while allowing to verify that the sum of outputs of a transaction is no more than the sum of inputs. Monero [Noe15] uses a special type of signature scheme to hide the origins and destinations of transactions among a set of UTXOs chosen by the sender (anonymity set). The size of the signature, however, increases linearly with the size of the anonymity set. Thus, the anonymity properties of the extension to Zether is similar to that Monero.

Zcash [zcaa], based on Zerocash [BCG$^+$14], provides anonymity at a sublinear cost using a more sophisticated ZK-proof system called zkSNARKs [GGPR13]. Senders and recipients are hidden among the group of people who use shielded addresses. Both Monero and ZCash utilize a set of nullifiers which grows linear in the number of transactions. The downside of using SNARKs is that a large common reference string (CRS) needs to be generated beforehand in a way that no one knows the trapdoor, which is a challenging task [pow]. Spenders needs to download the CRS and generate proofs for a large circuit, which is very time consuming [BCG$^+$14, zcab].

CoinJoin [Max13] provides a way for a set of users to jointly create a Bitcoin transaction. MimbleWimble/Grin [Poe16, gri] combines confidential transactions [Max15] and CoinJoin along with techniques to aggregate transactions non-interactively. CoinShuffle [RMK14] and Mixcoin [BNM$^+$14] are mixing protocols for Bitcoin. TumbleBit [HAB$^+$17] uses an untrusted intermediary, called a tumbler, to make transactions unlinkable. Möbius [MM18] replaces the tumbler with an Ethereum smart contract. Zether's approach to anonymity is different from the above: it does not rely on *active* participation of other users. Zether users can choose their own anonymity set like Monero. On the other hand, if a mixing service is used actively, it may provide better anonymity.[4]

Hawk [KMS$^+$16] is a framework for building arbitrary smart contracts in a privacy-preserving way. In particular, it can completely hide the bid values in an auction. This generality, however, comes at a significant cost. In Hawk, the private portion of a contract is converted into a circuit. A manager, who is trusted with the private inputs of participants, generates a zkSNARK proof on the circuit [BSCTV14] to show that it has been executed correctly. Apart from the fact that SNARKs rely on trusted setup, the reference string is also circuit-dependent, so a different string needs to be generated for every contract. Moreover, the circuit model puts a bound on the number of users who can participate.

As a result, though Hawk is quite powerful and could provide better privacy, it is not fully decentralized and would be too expensive to use for simple contracts. Another general-purpose framework, Ekiden [**?**], addresses both the performance and confidentiality problems with smart contract platforms, but relies on trusted execution environments like Intel SGX, so are not fully decentralized either.

[4] One can potentially use Zether in combination with Möbius on Ethereum to get the best of both worlds. We leave this as an interesting open question.

RSCoin, Solidus, zkLedger, etc. [DM16, CZJ+17, NVV18] operate in a model that falls somewhere between a fully decentralized setting like that of Bitcoin/Ethereum and a centralized setting like that of modern financial systems. In this model, the banks regulate the monetary supply but use a blockchain to transact. There is some similarity between the techniques used here and zkLedger's, where every bank has an account. A sending bank A in zkLedger creates several commitments to send some money x to a receiving bank B. The commitment corresponding to A is to $-x$, to B is to x, and all other commitments are to zero. Then, there are proofs to show that the commitments are well-formed and A has more than x amount of money. While we use similar ideas in our protocol, Zether needs to deal with issues like front-running, replay, compatibility, etc. that come with building a smart contract on an open platform.

Concurrent Work. Zexe [BCG+18] is a recent proposal for a private scripting language for Zerocash-style currencies. It provides similar functionality to Bitcoin script while hiding the inputs to the script and the script itself. It, however, does not support stateful computations in the way a smart contract does.

QuisQuis [FMMO18] is a new anonymity system designed to address some of the problems with cryptocurrencies like Monero and Zcash (e.g., the set of unspent outputs keep growing). Their model is an interesting hybrid of UTXO and account models. While the basic unit is an account (consisting of a public key and a commitment), they are only of one-time use: old accounts are destroyed and new accounts created in a transaction. Our Σ-Bullets protocol is similar to the techniques used in QuisQuis [FMMO18], where a Pedersen commitment contains the same value as an ElGamal encryption and then execute the Bulletproof on the ElGamal encrypted values. Σ-Bullets more directly incorporates the Σ-protocol with the Bulletproof protocol.

Unfortunately, QuisQuis suffers from front-running attacks (public keys in an anonymity set may get updated just before the transaction is processed) and puts additional burden on clients (they have to go through the list of all updated keys to find out which one belongs to them). More importantly, QuisQuis is a standalone cryptocurrency while Zether is a system that can be deployed on any smart contract platform, and can be used by other smart contracts to achieve privacy.

4 The Zether Protocol

Notations. We use λ to denote the security parameter. Let $\mathsf{GroupGen}$ be a polynomial-time algorithm that on input 1^λ outputs $(p,g,\mathbb{G})$ where $p = \Theta(\lambda)$, p is prime, $\mathbb{G}$ is a group of order p, g is a generator of $\mathbb{G}$, and the decisional Diffie-Hellman (DDH) assumption holds in $\mathbb{G}$. The DDH assumption states that a tuple $(g,g^a,g^b,g^{a\cdot b})$ is computationally indistinguishable from (g,g^a,g^b,g^c) for random a,b,c. It implies the discrete logarithm assumption.

Let $\mathbb{Z}_p$ denote the integers modulo p. $\mathbb{Z}_p^\star$ is the set of inverses in $\mathbb{Z}_p$. We use $[a,b]$ for $a,b \in \mathbb{Z}$ to denote the set of integers $\{a,a+1,...,b-1,b\}$. We use $x \leftarrow_\$ S$ to denote that x is sampled uniformly at random from a set S. We use PPT as a shorthand for probabilistic polynomial time and $\mathsf{negl}\,(\lambda)$ to denote negligible functions.

In Appendix B, we define the cryptographic primitives used by Zether, namely ElGamal encryption, zero-knowledge proofs, and digital signatures.

Zether Components. The Zether consists of three components: a global setup algorithm that is run once to generate the global parameters for the protocol as well as to deploy the Zether smart contract. The second component is the Zether smart contract (ZSC) that handles transactions between users, interoperability with external smart contracts, and keeps the state of the system. The final component of the mechanism are the user algorithms which describe how users can interact with the smart contract and create valid transactions. A user is of course not bound to the behavior described in the user algorithms. Our security proof in Appendix D shows that even if an adversarial user does not comply with these algorithms, he can't break Zether's correctness, privacy and over-draft protections.

Setup. The setup algorithm calls $\mathsf{Setup}_{\mathsf{nizk}}$ and $\mathsf{Setup}_{\mathsf{nizk}}$ as subroutines which are the setup algorithms for the proof system and the signature scheme, respectively. The former setup could depend on the relations for which proofs are constructed. If these subroutines are trustless, then the whole setup is trustless, meaning that its correctness can be verified publicly. In the implementation (Section 5), we use Bulletproofs [BBB+18] and Schnorr signatures [Sch90], both of which have a trustless setup. Zether significantly differs from Zcash [zcaa] in this respect because Zcash has a trusted setup and its security is broken if the setup is subverted.

Setup algorithm is formally described in Figure 2. Apart from setting up the proof system and signature scheme, it initializes account tables acc and pending transfers table $\mathsf{pTransfers}$ (recall that incoming transfers are put into a pending state first), a last roll over epoch table $\mathsf{lastRollOver}$ to keep track of the last epochs accounts were updated, a lock table lock to keep track of the addresses to which accounts are locked, a counter table ctr to prevent replay attacks, and a variable b_{total} that tracks the total amount of ZTH held by the contract. The setup also specifies an epoch length E and a maximum amount value MAX.

Zero-Knowledge Relations. Each transfer and burn transaction in Zether contains a ZK-proof which ensures that the transfer is valid without revealing the reasons why it is valid.

Burn Transaction. Let us first consider a burn transaction where a user needs to verifiably decrypt his Zether balance. It can certainly do this by revealing its secret key to the smart contract. However, an adversary can use the secret key to decrypt all previous balances and transactions of the user, thus completely breaking its privacy. So, instead of decrypting in the clear, the user creates a ZK-proof for the following statement:

$$\mathsf{st}_{\mathsf{burn}} : \left\{ (y, C_L, C_R, u, b, g, g_{\mathsf{epoch}}; \mathsf{sk}) : y = g^{\mathsf{sk}} \wedge C_L = g^b C_R^{\mathsf{sk}} \right\}. \tag{1}$$

The statement shows that the user knows an sk such that y is indeed the public key corresponding to sk and (C_L, C_R) is a valid encryption of b under y. A simple Σ-protocol can be used to prove the statement.

Transfer Transaction. Let us now consider a transfer transaction. Suppose a user wants to transfer an amount $b^\star$ from a public key y to a public key $\overline{y}$. Let (C_L, C_R)

be the encryption of balance associated with y. The smart contract needs to deduct $b^\star$ from y's balance and add the same amount to $\overline{y}$'s balance, which will be put into a pending state. Since we need to hide $b^\star$ in this process, user will encrypt $b^\star$ under both y and $\overline{y}$ to get (C,D) and $(\overline{C},\overline{D})$, respectively. Now, it must provide a proof to show that:

1. both ciphertexts are well formed and encrypt the same value $b^\star$;
2. $b^\star$ is a positive value; and,
3. the remaining balance of y, say b', is positive too.

More formally, a user proves the following statement:

$$\begin{aligned}\mathsf{st}_{\mathsf{ConfTransfer}} : \{(y,\overline{y},C_L,C_R,C,D,\overline{C},g;\mathsf{sk},b^\star,b',r) : \\ C = g^{b^\star} y^r \wedge \overline{C} = g^{b^\star} \overline{y}^r \wedge D = g^r \wedge \\ C_L/C = g^{b'} (C_R/D)^{\mathsf{sk}} \wedge y = g^{\mathsf{sk}} \wedge \\ b^\star \in [0,\mathsf{MAX}] \wedge b' \in [0,\mathsf{MAX}]\}. \quad (2)\end{aligned}$$

Kurosawa [Kur02] first showed that in the ElGamal encryption scheme, randomness can be reused to encrypt to multiple recipients. We use the same idea here to make the zero-knowledge component more efficient: the same random number r is used to encrypt $b^\star$ under both y and $\overline{y}$.

Zether Contract. The Zether contract (ZSC) is defined in Figure 1. It consists of five public methods Fund, Burn, Transfer, Lock, Unlock and two additional internal helper methods RollOver, CheckLock. The helper methods are used to modularize the contract's logic. We use Solidity syntax at some places in the description of ZSC, instead of introducing new notation. We now discuss ZSC's methods in detail.

Rolling over. Pending transfers for an account must be rolled over into the account every epoch, or at least in the epochs the account is used. However, no instruction on a smart contract can execute unless triggered by a transaction. As a result, all public methods of ZSC first call RollOver on the input public key(s).

Given a public key y, RollOver checks if the last roll over was in an older epoch. If yes, then it rolls over the pending transfers pTransfers$[y]$ into acc$[y]$ and resets pending transfers as well as the last roll over epoch.

Check lock. Every transaction to operate on an account is associated with an Ethereum address (returned by msg.sender). If the account is unlocked, then it can be operated from any address. However, if it is locked to a certain address, then it can only be operated from that address. CheckLock is an internal methods to check these two conditions. All the methods call CheckLock before operating on an account.

Locking. Given a public key y, an address addr and a signature σ_{lock}, Lock checks if it is appropriate to operate on the account by calling CheckLock, which will be discussed in more detail shortly, and verifies that σ_{lock} is a valid signature on addr and the current value of counter ctr$[y]$. It sets lock$[y]$ to be addr and increments the counter, which ensures that this lock transaction cannot be replayed. Unlock method also calls CheckLock first, then sets the pending lock to be $\perp$.

Funding. Anybody can fund an account, even an account that he/she does not own, by simply specifying the public key y and transferring some ETH. The only

exception is for locked accounts; they can only be operated from the locking address. (One could have a different rule for funding locked accounts.) Fund converts ETH into ZTH. The ETH gets stored in the smart contract and the ZTH are homomorphically added to y's (pending) balance. If the account does not exist yet, a new one is created. Fund also ensures that the deposit does not exceed the total amount of funds, MAX, that Zether can handle.

Burn. Burn converts ZTH back to ETH. It verifies the proof π_{burn} against $\mathsf{st}_{\mathsf{burn}}$ (see (1)) to ensure that the sender knows the right private key and is claiming the right amount. It also checks a signature on the transaction data and the current value of counter, which prevents replay attacks. Note that a burn operation does not close an account.

Transfer. Transfer transfers some ZTH from an account to another. The proof π_{transfer} makes sure that the ciphertext has the right form and the sender has enough money (see (2)). Similar to Burn, there is a signature here to prevent replay attacks.

Note that the transferred amount is added to pTransfers of the recipient, not acc. (It will be rolled over into acc in a later epoch.) Thus, outgoing transfers of the recipient in this epoch will not be invalidated.

User Algorithms. User algorithms specify how users can interact with ZSC. CreateTransferTx and CreateBurnTx first do a roll over of the input public keys to ensure that any pending transfers are rolled over. CreateBurnTx uses ReadBalance to recover the amount of ZTH in the account. Using the private key, ReadBalance finds the right b s.t. $C_L/C_R^x = g^b$. In typical cases, a user would *not* have to try all positive integers one by one to recover b. She will already have a good estimate of b.

4.1 Anonymous Zether

We now describe the anonymous version of Zether. While this version hides both sender and receiver apart from hiding the transfer amount, it also incurs some additional costs. First, the size of ZK-proof for a transfer increases linearly with the size of the anonymity set. Second, as we will see, users would be able to do only one transfer or burn transaction per epoch (not one of each). We discuss some issues pertinent to the design of anonymous Zether below. For a detailed description of anonymous Zether, we refer the reader to the full version of this paper [Ano].

Replay and Double-Spend Protection. An anonymous transaction published by Alice involves multiple accounts only one of which Alice may own. To preserve anonymity, all the accounts involved in the transaction must be treated in the same way. Thus, the nonces associated with each one of them should be incremented. Other account holders involved in Alice's transaction may have generated a transaction with the previous value of nonce. Unfortunately, if their transactions get in later, then they will be rejected. If even one of them gets in before, then Alice's transaction will be rejected.

We take a different approach to replay protection, which has some similarities with that of Monero. Every epoch will be associated with a base g_{epoch} derived from hashing some fixed string like 'Zether' and the current epoch number. To initiate a transfer or burn transaction from an account with public key $y = g^{\mathsf{sk}}$,

$g_{\mathsf{epoch}}^{\mathsf{sk}}$ must be included in the transaction. More precisely, the proof π described above for a transfer transaction will also show knowledge of sk such that $\overline{g} = g_{\mathsf{epoch}}^{\mathsf{sk}}$ for $\overline{g}$ included in the transaction. (Burn transactions' proofs will also include this.) Importantly, $\overline{g}$ is computationally unlinkable to y under the DDH assumption. We refer to $\overline{g}$ as a nonce in the sequel.

While in the case of confidential transfers, we subtract the transfer amount from the sender's balance immediately but keep it pending for the receiver, one cannot take the same approach for anonymous transfers. All the transfer amounts, whether positive (for the receiver), negative (for the sender), or zero (for others) have to be kept pending. Thus, an anonymous transaction would not immediately affect the balance of any of the users involved. This opens up the system to double-spending attacks. A user could generate two transactions in an epoch, sending her total balance to two different users. The attached ZK-proofs would both be valid because they will be verified against the same state. Fortunately, the nonce, in addition to preventing replay attacks, also prevents such double-spending attacks.

During every epoch, ZSC will accumulate nonces as they come, rejecting any transaction that reuses a nonce. An important difference from Monero is that the set of nonces does not grow indefinitely; it is reset to null at the beginning of every epoch. Thus, providing anonymity does not lead to a continuous growth in the size of the state of ZSC. A drawback of this approach to replay protection and double-spending is that even honest users can only initiate at most one transfer or burn transaction in a given epoch.

Global Updates. With the new replay protection mechanism in place, a few global updates need to be made in every epoch: set the base for the epoch and empty the nonce set. We will have to make the updates at the receipt of the very first message in an epoch, be it from any account. Thus, users will have to provide a little more gas to cover the possibility that their message could be the very first one in an epoch. In most cases, this extra gas will be reimbursed.

Locking to Smart Contracts. If some accounts involved in an anonymous transfer are locked to a smart contract, then all of the locked accounts must be locked to the same contract. Furthermore, the transfer is processed only if it comes from that contract. Also, locking must not come into effect immediately. Suppose Alice publishes a transaction in a certain epoch to lock her account to a smart contract. Another user Bob may have published a transfer transaction (at about the same time as Alice) with Alice in his anonymity set while her account was still unlocked. If Alice's transaction gets in first, locking her account, then Bob's transaction will be rejected. The same holds for unlocking as well. Therefore, when ZSC is invoked to lock/unlock an account, it just records the request but does not act on it immediately. When the account is rolled over in some later epoch, the request will be executed.

Lock transactions also need replay protection. In fact, using the account secret key, the sender must sign both the nonce and an address (to which the account will be locked) in the case of confidential transfers, and both the epoch base and address in the case of anonymous transfers. As a result, for the latter case, lock transactions must be published at the beginning of an epoch just like transfer and burn transactions.

4.2 Σ-Bullets

Transfer and AnonTransfer are relatively-large relations that involve proofs on encrypted data. We, therefore, want to use a proof system that (1) is efficient, i.e., has short proofs and efficient verification, and (2) allows proofs on encrypted data. Bulletproofs [BBB+18] is a generic zero-knowledge proof system that produces short (logarithmic sized) proofs without relying on a trusted setup. Bulletproofs was specifically designed to work with Confidential Transactions (CT) [Max15] as it directly proves statements containing values committed to as Pedersen commitments. Its short proofs and trustless setup make Bulletproofs an intriguing choice for Zether's underlying proof system. However, unlike the UTXO-based CT, Zether relies on ElGamal encryptions as commitments. We, therefore, aim to use a proof system that can prove statements on ElGamal ciphertexts.

It is not sufficient to simply replace Pedersen commitments with ElGamal encryptions as the latter cannot be "opened" similar to commitments and are also not additively homomorphic if encryptions are under different keys, as is the case in Zether. Also, for AnonTransfer, we need to combine a one-out-of-many proof[5] with range proofs. A one-out-of-many proof is used to select the receiver and sender transfer encryption and the range proof ensures that no overdraft happens. Bulletproofs enables efficient range proofs and there are logarithmic sized efficient Σ-protocols [CD98] for doing one-out-of-many proofs [GK15, BCC+15].

To efficiently prove these statements and instantiate Zether, we design Σ-Bullets as an extension of Bulletproofs. Given an arithmetic circuit, a Σ-Bullets proof ensures that a public linear combination of the circuit's wires is equal to some witness of a Σ-protocol. This enhancement in turn enables proofs on many different encodings such as ElGamal encryptions, ElGamal commitments, or Pedersen commitments in different groups or using different generators. Further, it allows the combination of different specialized Σ-protocols such as one-out-of-many proofs or accumulator proofs [CL02] with the generic circuit-base proof system Bulletproofs. This will benefit other systems that want to prove statements on additively-encoded witnesses.

We describe Σ-Bullets in detail in Appendix E.

5 Empirical Evaluation

We implemented basic Zether as an Ethereum smart contract showing that Zether is feasible today and can be run on top of the Ethereum virtual machine. We also discuss several optimizations that we made in order to improve the performance of the contract. Further, we will analyze what small improvements to the EVM would significantly benefit Zether. Some of these improvements have been discussed independently and are already part of the Ethereum improvement proposal (EIP) track.

5.1 Solidity Implementation & Optimizations

The Zether smart contract is implemented in Solidity and makes use of several observations. Ethereum recently introduced precompiled contracts for elliptic-curve operations on the curve BN-128 [bn1a]. These precompiled contracts reduce the

[5] A non-interactive one-out-of-many proof can be used to instantiate a ring-signature in which a signer reveals that she knows a private key out of

cost of executing these operations compared to direct implementations. The reason is that miners can use specialized software, i.e., special cryptography libraries, to run these functions more efficiently. The operations were originally introduced to support pairing-based ZK-SNARKs. Σ-Bullets do not require pairings and the curve BN-128 is not an optimal choice in terms of efficiency or security for Bulletproofs/Σ-Bullets. Nevertheless, we chose to implement Zether using this curve because it is natively supported (precompiled contracts are far cheaper than a Solidity implementation of another curve such as secp256k1 [sec].) As we explain in Section 5.3, this means that we have to rely on the DDH assumption in the $\mathbb{G}_1$ group of BN-128. This assumption is called the external DDH or XDH assumption and is less general than the DDH assumption.

Despite the precompiled contract, a majority of the gas cost lies in the cryptographic operations used, especially curve multiplication. We therefore aimed to reduce the number of exponentiations to an absolute minimum. We did this by implementing the optimizations presented in Section 6.2 of [BBB+18].

We did not implement multi-exponentiation as this would not be beneficial. Multi-exponentiations reduce the number of curve operations but do this by splitting up the exponentiation. Multi-exponentiation algorithms assume that a k-bit exponentiation use k curve operations. This is not the case for Solidity however. The gas cost for an exponentiation is independent of the exponents length and curve additions are relatively overpriced to curve multiplications. A curve multiplication is only 80 times more expensive than a curve addition even if the exponent has 256 bits. Therefore, multi-exponentiation would not lower but increase the gas cost.

In a further optimization, we rolled out the inner product argument and combined all possible exponentiations into a single large statement. Furthermore, we slightly modified the recursive inner product argument such that it terminates at $n = 4$ instead of $n = 1$. By doing this, the prover has to send 6 more elements in $\mathbb{Z}_p$ but on the other hand saves sending 4 Pedersen hashes which are elements in $\mathbb{G}$. Since Solidity does not support point compression, i.e., points in $\mathbb{G}$ are encoded using 64 bytes and scalars using 32 bytes, this small modification therefore saves 64 bytes in space and also reduces the number of curve exponentiations that need to be done. In total for the ConfTransfer transaction, the elliptic curve operations for the account state manipulations, the Σ-protocol and the 2 32-bit range proofs use 156 curve additions and 154 curve multiplications.

A further optimization concerns the common reference string (CRS). Bulletproofs unlike SNARKs do not use a structured reference string which would require a trusted setup. Nevertheless, Bulletproofs still requires a long linear-sized reference string that the verifier needs to access. While the CRS could be generated on the fly, this would add over 3.9 million gas to the cost of the transaction. Storing the CRS in the blockchain storage also creates high additional cost as loading a 32-byte word costs 200 gas. On the other hand, loading a 32-byte code instruction costs only 3 gas which is why we choose to hard-code the generators into the smart contract. While this makes the contract-generation process more expensive, it is a one-time cost which is amortized over the lifetime of a contract.

	Gas Cost	in $	EC Cost	|tx|
Burn	384k	0.080	329k	160 bytes
Fund	260k	0.035	41k	64 bytes
Transfer	7,188k	1.51	6,455k	1,472 bytes
Lock	223k	0.049	83k	128 bytes
Unlock	193k	0.041	83k	96 bytes

Table 1. Gas costs of ZSC methods

5.2 Measurements

We now present several measurements for our implementation of basic Zether. We measure the total gas cost which includes the basic cost for sending a transaction, the storage cost as well as the proof/signature verification. We also present the gas cost in USD using a gas cost price of 2 Gwei per unit of gas [Ethb] and exchange rate of 105 USD per ETH [mar]. At the time of writing, a basic Zether transaction costs about 1.5 USD. We also show that a majority of the cost is produced by elliptic-curve operations by factoring out their gas cost. For a transfer transaction, the elliptic-curve operations make up 90% of the total cost. For a fund transaction, the majority of the cost comes from initializing a new account. Adding funds to an existing account is significantly cheaper. Finally, we present the size of the transaction data. Note that this does not include the basic Ethereum transaction data which is roughly 110 bytes.

5.3 Ethereum Limitations & Future Directions

Currently, Ethereum's computation power is very limited. A simplified estimate is that at 3 gas units per arithmetic operation, Ethereum currently supports less than 180k operations per second for the whole network. There are several efforts to increase the scalability of Ethereum [BG17, Zam17]. The majority of the cost of a transaction in Zether comes from the cryptographic operations. Despite heavily optimizing them, they make up for almost 90% of the cost. These operations seem overpriced when compared to operations like hashing. This discrepancy has been noted and discussed independently [bn1b]. There currently exists an EIP to reduce the gas cost of elliptic curve multiplications by a factor of 6.66 and additions by a factor of 3.33 [bn1b]. A further EIP reduces the cost of calling a precompiled contract [pre] which would reduce the cost for each cryptographic operation by another 700 units of gas. If both of these were implemented, the cost of a Zether transfer would reduce to roughly 1.7 million gas (0.36 USD). At that point, optimizations on the non-cryptographic part of the contract could probably further reduce the cost.

There are further changes that Ethereum could make that would benefit Zether. One of them would be supporting elliptic-curve operations for more efficient curves like secp256k1 [sec] or Curve25519-ristretto [cur]. Another would be supporting multi-exponentiation techniques that can reduce the number of cryptographic operations needed to verify the range proofs [Pip80].

A simple but significant optimization that can be implemented without changing Ethereum applies to the proof verification: Bulletproofs can be batch verified. This means that verifying k proofs is significantly faster than verifying a single proof. If transactions were collected by some service provider, combined

to a single transaction and then sent to the Zether contract, it would significantly reduce the verification cost per proof. However, all transactions in a batch must be valid because a single invalid transaction will cause the whole verification to fail. Batch verification requires randomness but this randomness can either be sampled from the block header [BCG15] or generated from a hash of the proofs.

6 Applications of Zether

We now discuss how Zether can be applied to "add" privacy seamlessly to important domains such as auctions, payment channels, and voting. For a more detailed discussion on each application, we refer the reader to the full version of this paper [Ano].

Sealed-Bid Auctions. Auctions are used to sell a wide variety of resources on Ethereum [Tok]. A primary example is the Ethereum Name Service (ENS) [ENS]. In the ENS bidding phase, a bidder submits a hash of her bid along with an amount of ether. ENS suggests that the amount should be higher than the bid value to disguise the true value of the bid. There are several drawbacks to this approach including that it reveals a good upper-bound to bid value. Using Zether, bidders can simply lock their accounts (or transfer the bid value to a new account and lock that) to the auction contract, thus getting full bid confidentiality. No other collateral needs to be put in place. After the bidding phase, bidders can open their bids by providing a ZK-proof on the encrypted bid value (a *burn* proof essentially). The auction smart contract can then simply unlock the accounts of the bidders who lost.

Confidential Payment Channels. Payment channels are widely considered to be one of the most important solutions to the scalability problem of cryptocurrencies. On Ethereum, they can be set up very easily through smart contracts. A certain amount of ether is locked into the contract, transfers are conducted off-the-chain between the parties, and then a final settlement is made on chain. This clearly reveals a lot of information, but Zether can be used to prevent it. A Zether account could be locked to the channel contract and ZK-proofs could be exchanged offline to show that the channel has enough liquidity.

Stake Voting. Several blockchain-based voting protocols (e.g., [MSH17]) have been proposed to deal with the transparency issues of traditional voting systems. Here, the vote values are binary: Participants publish a special encryption of their votes and prove that they are 0 or 1. With Zether, we can allow the votes to carry weights, proportional to the assets a participant owns. Once again, this can be achieved by just locking account to the vote contract and proving in zero-knowledge that the vote value is equal to the amount locked.

Proof-of-Stake Consensus. Proof-of-stake is a popular alternative to the wasteful proof-of-work consensus mechanism. Users *stake* a number of coins and then a random beacon is used to select one of them as leader. This reveals the stake of users, making them susceptible to targeted attacks. Zether could be used to make proof-of-stake confidential. At a high level, users could encrypt an initial lottery ticket t under their public key and stake an encrypted Zether balance b under the same key. Then the random beacon value is used to derive a lottery drawing v. If v falls between t and $t+b$ then the user wins the lottery, which it can prove without revealing t or b.

References

AABN02. Michel Abdalla, Jee Hea An, Mihir Bellare, and Chanathip Namprempre. From identification to signatures via the Fiat-Shamir transform: Minimizing assumptions for security and forward-security. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 418–433. Springer, April / May 2002.

Ano. Anonymized. Zether: Towards privacy in a smart contract world. Provided with permission from PC-chairs, SHA224: 9f643e51be10f2bff0aa4b96d1cba615bee7116ba4b30ec1b284fc22.

BAZB19. Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. Cryptology ePrint Archive, Report 2019/191, 2019. https://eprint.iacr.org/2019/191.

BBB+18. Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334. IEEE Computer Society Press, May 2018.

BCC+15. Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Essam Ghadafi, Jens Groth, and Christophe Petit. Short accountable ring signatures based on DDH. Cryptology ePrint Archive, Report 2015/643, 2015. http://eprint.iacr.org/2015/643.

BCC+16. Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 327–357. Springer, May 2016.

BCG+14. Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. Cryptology ePrint Archive, Report 2014/349, 2014. http://eprint.iacr.org/2014/349.

BCG15. Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. On bitcoin as a public randomness source. Cryptology ePrint Archive, Report 2015/1015, 2015. http://eprint.iacr.org/2015/1015.

BCG+18. Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. Cryptology ePrint Archive, Report 2018/962, 2018. https://eprint.iacr.org/2018/962.

BCS16. Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. Cryptology ePrint Archive, Report 2016/116, 2016. http://eprint.iacr.org/2016/116.

BG17. Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.

bn1a. Precompiled contracts for addition and scalar multiplication on the elliptic curve alt bn128. https://eips.ethereum.org/EIPS/eip-196.

bn1b. Reduce alt bn128 precompile gas costs. https://eips.ethereum.org/EIPS/eip-1108.

BNM+14. Joseph Bonneau, Arvind Narayanan, Andrew Miller, Jeremy Clark, Joshua A. Kroll, and Edward W. Felten. Mixcoin: Anonymity for Bitcoin with accountable mixes. Cryptology ePrint Archive, Report 2014/077, 2014. http://eprint.iacr.org/2014/077.

BS18. Dan Boneh and Victor Shoup. *A graduate course in applied cryptography*. Cambridge, 2018.

BSCTV14. Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 781–796. USENIX Association, 2014.

But16. Vitalik Buterin. Thoughts on UTXOs. https://medium.com/@ConsenSys/thoughts-on-utxo-by-vitalik-buterin-2bb782c67e53, 2016.

CD98. Ronald Cramer and Ivan Damgård. Zero-knowledge proofs for finite field arithmetic; or: Can zero-knowledge be for free? In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 424–441. Springer, August 1998.

CGS97. Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 103–118. Springer, May 1997.

CL02. Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 61–76. Springer, August 2002.

cur. Curve25519-ristretto. https://ristretto.group/.

CZJ+17. Ethan Cecchetti, Fan Zhang, Yan Ji, Ahmed E. Kosba, Ari Juels, and Elaine Shi. Solidus: Confidential distributed ledger transactions via PVORM. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17*, pages 701–717. ACM Press, October / November 2017.

CZK+18. Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah M. Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contract execution. *CoRR*, abs/1804.05141, 2018.

Dåm. Ivan Dåmgard. On sigma protocols. https://www.cs.au.dk/~ivan/Sigma.pdf.

DM16. George Danezis and Sarah Meiklejohn. Centrally banked cryptocurrencies. In *NDSS 2016*. The Internet Society, February 2016.

ENS. Ethereum Name Service. https://ens.domains.

etha. Ethereum Project: Blockchain App Platform. https://www.ethereum.org/.

Ethb. Ethereum Gasstation. https://ethgasstation.info/calculatorTxV.php.

FMMO18. Prastudy Fauzi, Sarah Meiklejohn, Rebekah Mercer, and Claudio Orlandi. Quisquis: A new design for anonymous cryptocurrencies. Cryptology ePrint Archive, Report 2018/990, 2018. https://eprint.iacr.org/2018/990.

FS87. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, August 1987.

GGPR13. Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 626–645. Springer, May 2013.

GK15. Jens Groth and Markulf Kohlweiss. One-out-of-many proofs: Or how to leak a secret and spend a coin. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 253–280. Springer, April 2015.

gri. Grin. https://grin-tech.org/.

HAB+17. Ethan Heilman, Leen Alshenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. TumbleBit: An untrusted bitcoin-compatible anonymous payment hub. In *NDSS 2017*. The Internet Society, February / March 2017.

KMS+16. Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-

preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy*, pages 839–858. IEEE Computer Society Press, May 2016.

Kur02. Kaoru Kurosawa. Multi-recipient public-key encryption with shortened ciphertext. In David Naccache and Pascal Paillier, editors, *PKC 2002*, volume 2274 of *LNCS*, pages 48–63. Springer, February 2002.

LCO+16. Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269. ACM, 2016.

lib19. The libra blockchain whitepaper. https://developers.libra.org/docs/assets/papers/the-libra-blockchain.pdf, 2019.

mar. Total Market Capitalization. https://coinmarketcap.com/charts.

Max13. Greg Maxwell. Coinjoin: Bitcoin privacy for the real world, 2013. https://bitcointalk.org/?topic=279249.

Max15. Greg Maxwell. Confidential transactions, 2015. https://people.xiph.org/~greg/confidential_values.txt.

MGGR13. Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed E-cash from Bitcoin. In *2013 IEEE Symposium on Security and Privacy*, pages 397–411. IEEE Computer Society Press, May 2013.

MM18. Sarah Meiklejohn and Rebekah Mercer. Möbius: Trustless tumbling for transaction privacy. *PoPETs*, 2018(2):105–121, 2018.

mov19. Move: A language with programmable resources. https://developers.libra.org/docs/assets/papers/libra-move-a-language-with-programmable-resources.pdf, 2019.

MSH17. Patrick McCorry, Siamak F. Shahandashti, and Feng Hao. A smart contract for boardroom voting with maximum voter privacy. Cryptology ePrint Archive, Report 2017/110, 2017. http://eprint.iacr.org/2017/110.

Noe15. Shen Noether. Ring signature confidential transactions for monero. Cryptology ePrint Archive, Report 2015/1098, 2015. http://eprint.iacr.org/2015/1098.

NVV18. Neha Narula, Willy Vasquez, and Madars Virza. zkledger: Privacy-preserving auditing for distributed ledgers. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pages 65–80, 2018.

Ped92. Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '91, pages 129–140. Springer-Verlag, 1992.

Pip80. Nicholas Pippenger. On the evaluation of powers and monomials. *SIAM Journal on Computing*, 9(2):230–250, 1980.

Poe16. Andrew Poelstra. Mimblewimble, 2016. https://scalingbitcoin.org/papers/mimblewimble.pdf.

pow. Announcing the world's largest multi-party computation ceremony. https://www.zfnd.org/blog/powers-of-tau/.

pre. PRECOMPILED CALL opcode (Remove CALL costs for precompiled contracts). https://eips.ethereum.org/EIPS/eip-1109.

RMK14. Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. CoinShuffle: Practical decentralized coin mixing for bitcoin. In Miroslaw Kutylowski and Jaideep Vaidya, editors, *ESORICS 2014, Part II*, volume 8713 of *LNCS*, pages 345–364. Springer, September 2014.

Sch90. Claus-Peter Schnorr. Efficient identification and signatures for smart cards (abstract) (rump session). In Jean-Jacques Quisquater and Joos Vandewalle, editors, *EUROCRYPT'89*, volume 434 of *LNCS*, pages 688–689. Springer, April 1990.
sec. Secp256k1. https://en.bitcoin.it/wiki/Secp256k1.
Sol. Solidity webpage. https://solidity.readthedocs.io.
Sza96. Nick Szabo. Smart contracts: Building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought*, 16, 1996.
Tok. Analyzing Token Sale Models. https://vitalik.ca/general/2017/06/09/sales.html.
Woo. Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. https://gavwood.com/paper.pdf.
Zam17. Vlad Zamfir. Casper the friendly ghost: A correct by construction blockchain consensus protocol. https://github.com/ethereum/research/blob/master/papers/CasperTFG/CasperTFG.pdf, 2017.
zcaa. Zcash: Privacy-protecting digital currency. https://z.cash/.
zcab. zcash Documentation. https://media.readthedocs.org/pdf/zcash/english-docs/zcash.pdf.

A Background on Ethereum

Accounts are Ethereum's basic units. There are (a) *externally-owned accounts (EOAs)*, controlled by private keys and (b) *contract accounts*, controlled by their code. Both types of accounts have an ether balance, denominated in units of *wei*: 1 ether is 1e18 wei. The Ethereum blockchain tracks the state of every account [etha, Woo]. State changes are initiated through *transactions* coming from EOAs. A transaction consists of the destination account address, a signature σ, the transferred amount in wei, an optional data field representing inputs to a contract, a gasLimit value, and a gasPrice value. Every EOA is associated with a nonce, a counter that increments with every transaction. The signature σ signs the transaction and the sender's nonce. During transaction processing, σ is verified against the nonce value. As a result, transactions cannot be "replayed" on the Ethereum network [etha].

A transaction can transfer wei between accounts or trigger the execution of smart contract code. Contracts can send messages to other contracts, mimicking function calls. Every transaction and code execution is replicated on all nodes in the network. Every executed operation has a specified cost expressed in terms of *gas* units. For example, storing 256 bits of data costs 20,000 units of gas while changing it costs 5,000 and reading it costs 200 [Woo]. The sender pays for all contract operations that the transaction calls.

The sender sets gasLimit field to the total amount of gas she is willing to spend for a transaction, and the gasPrice field to the amount of wei she is willing to pay per unit of gas. A miner, that is happy with the gas price, can include the transaction in a block and collect the fee. If the gas limit falls short of the gas needed to process the transaction, the miner will collect the fee but not change the blockchain's state. Excess fees are refunded to the account that issued the transaction [Woo].

The total gas consumed by all transactions in a block is limited. This ensures that the time needed for processing and propagating a block remains sufficiently small, allowing for an adequately-decentralized network. Currently that limit is around

8 million gas units. Simple arithmetic operations cost 3 gas units and the average block time is 15s. The total Ethereum network can, therefore, perform less than 180k arithmetic operations per second. Some complex operations, e.g., the Keccak 256-bit hash function, however, do not need to be arithmetized but are provide as a standalone functionality at a reduced cost (36 gas for a 32 byte hash) [Woo].

Contracts are written in specific programming languages such as Solidity [Sol]. Once compiled to bytecode, the contract can be read and executed by the *Ethereum virtual machine (EVM)*, a sandboxed and isolated run-time environment. The EVM has access to a global persistent storage system and each contract account has separate storage available to it.

In Ethereum, transactions are processed individually in an arbitrary order. Therefore, it is important to ensure that contract codes are written properly so that unexpected outcomes are avoided when a common part of the EVM state is changed by two or more transactions [LCO+16]. The low computational power, along with the asynchronous transactional nature of the Ethereum network make programming complicated smart contracts a delicate endeavor.

B Preliminaries

ElGamal Encryption. ElGamal encryption is a public key encryption scheme secure under the DDH assumption. A random number from $\mathbb{Z}_p^\star$, say x, acts as a private key, and $y = g^x$ is the public key corresponding to that. To encrypt an integer b, it is first mapped to one or more group elements. If $b \in \mathbb{Z}_p$, then a simple mapping would be to just raise g to b. Now, a ciphertext for b is given by $(g^b y^r, g^r)$ where $r \leftarrow_\$ \mathbb{Z}_p^\star$. With knowledge of x, one can divide $g^b y^r$ by $(g^r)^x$ to recover g^b. However, g^b needs to be brute-forced to compute b.

We argue that this is not an issue. First, as we will see, the Zether smart contract does not need to do this, only the users would do it. Second, users will have a good estimate of ZTH in their accounts because, typically, the transfer amount is known to the receiver. Thus, brute-force computation would occur only rarely. Third, one could represent a large range of values in terms of smaller ranges. For instance, if we want to allow amounts up to 64 bits, we could instead have 2 amounts of 32 bits each, and encrypt each one of them separately. In this paper, for simplicity, we will work with a single range, 1 to MAX, and set MAX to be 2^{32} in the implementation.

The primary benefit of putting balances in exponent is that it makes ElGamal encryption additively homomorphic. If b and b' are encrypted under the same public key y to get ciphertexts $(C_L = g^b y^r, C_R = g^r)$ and $(C'_L = g^{b'} y^{r'}, C'_R = g^{r'})$ respectively, then $(C_L C'_L = g^{b+b'} y^{r+r'}, C_R C'_R = g^{r+r'})$ is an encryption of $b+b'$ under y.

Zero-Knowledge Proofs. A zero-knowledge (ZK) proof of a statement does not reveal any information beyond the validity of the statement. For example, one could prove that two ciphertexts encrypt the same message without revealing the message itself. Though any NP statement can be proved in zero-knowledge, the concrete costs depend on a number of factors.

Σ-protocols are honest-verifier public-coin zero-knowledge interactive proofs of a special form. Very efficient Σ protocols exist for proving a wide variety of

algebraic statements like knowledge of b and r s.t. an ElGamal ciphertext encrypts b with randomness r. The Fiat-Shamir transform is a way of transforming any public-coin honest-verifier ZK-proof (like Σ protocols) into a *non-interactive* zero-knowledge *proof of knowledge* in the random oracle model.

A ZK-proof for the statement

$$\mathsf{st} : \{(a,b,c,...;x,y,z,...) : f(a,b,c,...,x,y,z,...)\}$$

means that the prover shows knowledge of $x,y,z,...$ such that $f(a,b,c,...,x,y,z,...)$ is true, where $a,b,c,...$ are public variables. We use $\mathsf{st}[a,b,c,...]$ to denote an instance of st where the variables $a,b,c,...$ have some fixed values.

We represent a non-interactive ZK (NIZK) proof system with algorithms $(\mathsf{Setup}_{\mathsf{nizk}},\mathsf{Prove},\mathsf{Verify}_{\mathsf{nizk}})$, where $\mathsf{Setup}_{\mathsf{nizk}}$ outputs some public parameters, Prove generates a proof for a statement given a witness, and $\mathsf{Verify}_{\mathsf{nizk}}$ checks if the proof is valid w.r.t the statement. Zether uses NIZKs that are a) correct, an honest prover can produce a valid proof b) zero-knowledge, a verifier learns nothing from the proof but the validity of the statement, and c) sound, a computationally bounded prover cannot convince a verifier of a false statement. Σ protocols, with the Fiat-Shamir transform applied, have all these properties.

Digital Signatures. Signature schemes are used to authorize messages by *signing* them. A verifier can check a signature but will be unable to forge a signature on a previously unsigned message. Signatures can be built from Fiat-Shamir transformed NIZK proofs [AABN02].

We represent a signature scheme with algorithms $(\mathsf{Setup}_{\mathsf{nizk}},\mathsf{Sign},\mathsf{Verify}_{\mathsf{nizk}})$, where $\mathsf{Setup}_{\mathsf{nizk}}$ outputs some public parameters, Sign generates a signature on an input message, and $\mathsf{Verify}_{\mathsf{nizk}}$ checks if the signature is valid w.r.t. the message. Zether requires a signature scheme that is a) correct, it is possible to create valid signatures on arbitrary messages and b) existentially unforgeable, a computationally bounded adversary can't create a valid signature on a *new* message, even after seeing signatures on other messages. We omit formal definitions for brevity and refer to [BS18] for a thorough treatment of the properties.

C Security Definition

A payment mechanism for Ethereum consists of a setup routine, user algorithms, and a smart contract. The contract maintains a state st which changes over time. The state at block height h is denoted by st_h. Users rely on the state of the smart contract to create transactions. A user account is identified by a public key pk. Let MAX be the maximum amount of money that the mechanism can handle. Any amount below must be an integer between 0 and MAX.[6]

The payment mechanism also provides a way to lock funds of an account to an Ethereum address addr so that the address can control the movement of funds through the account, until the lock is released. In Section 6, we will use the locking/unlocking feature to bring privacy to several commonly used smart contracts.

The term *transaction* is used in Ethereum to refer to a signed data package that stores a message to be sent from an externally owned account to another

[6] MAX should be treated as a constant with respect to the security parameter.

account on the blockchain. It can include binary data (its payload) and Ether. If the target account contains code, that code is executed and the payload is provided as input data. For a transaction tx, we use $\mathsf{tx.ETH}$ to denote the amount of ether being sent through tx.

Contracts can call other contracts or send Ether to non-contract accounts by the means of *message calls*. The message call most relevant to us is $\mathsf{msg.sender.transfer}$ (in the syntax of Solidity) which transfers a certain amount of Ether from a contract to the sender who called it. For a transaction tx that calls a function f on a contract, we use $f(\mathsf{tx}).\mathsf{ETH}$ to denote the amount of Ether successfully sent back to the caller when f is executed with tx (in Figure 1, where we describe the details of our Zether instantiation, we use a programming oriented, Solidity-inspired syntax). We are now ready to describe the various components of a payment mechanism.

Setup. In the setup phase, some public parameters are generated. They could be distributed off-chain or put into the smart contract (described below). The smart contract is also deployed.

User Algorithms. A user can run one of the following algorithms to interact with the smart contract. The output of these algorithms are *raw* transactions. We leave it implicit that they will be signed (using the public key of the Ethereum account from which they are sent) and destined to the Zether smart contract. Nonetheless, we use $\mathsf{tx.addr}$ to denote the Ethereum address addr from which tx was sent. All the algorithms get the security parameter as input but we show it explicitly only for the first one.

1. $\mathsf{CreateAddress}(1^\lambda) \to (\mathsf{sk}, \mathsf{pk})$. $\mathsf{CreateAddress}$ provides a way for a user to uniquely identify itself to the smart contract. It takes (a unary representation of) the security parameter as input and outputs a secret key sk and a public key pk. We assume that pk is derived in a deterministic way from sk, and use $\mathsf{pkOf}(\mathsf{sk})$ to denote the public key that corresponds to sk.
2. $\mathsf{CreateFundTx}(\mathsf{pk}, \mathsf{amt}) \to \mathsf{tx}_{\mathsf{fund}}$. $\mathsf{CreateFundTx}$ is used to add funds to an account. It takes a public key pk and an amount amt as inputs. It outputs $\mathsf{tx}_{\mathsf{fund}} = (\mathsf{pk},...)$.
3. $\mathsf{CreateTransferTx}(\mathsf{sk}_{\mathsf{from}}, \mathsf{pk}_{\mathsf{to}}, \mathsf{amt}, \mathsf{st}_h) \to \mathsf{tx}_{\mathsf{trans}}$. $\mathsf{CreateTransferTx}$ is used to transfer money from one account to another. It takes a secret key $\mathsf{sk}_{\mathsf{from}}$, a destination public key $\mathsf{pk}_{\mathsf{to}}$, an amount amt, and the state of the smart contract st_h at a certain block height h as inputs. It outputs $\mathsf{tx}_{\mathsf{trans}}$. (For anonymous transfers, this algorithm would also take a set $\mathsf{AnonSet}$ as input, which would contain both $\mathsf{pkOf}(\mathsf{sk}_{\mathsf{from}})$ and $\mathsf{pk}_{\mathsf{to}}$. $\mathsf{AnonSet}$ would be a part of the output too.)
4. $\mathsf{CreateBurnTx}(\mathsf{sk}, \mathsf{st}_h) \to \mathsf{tx}_{\mathsf{burn}}$. $\mathsf{CreateBurnTx}$ is used to withdraw the entire balance from an account. It takes a secret key sk and a state st_h as inputs. It outputs $\mathsf{tx}_{\mathsf{burn}} = (\mathsf{pkOf}(\mathsf{sk}), \mathsf{amt}, ...)$.
5. $\mathsf{CreateLockTx}(\mathsf{sk}, \mathsf{addr}, \mathsf{st}_h) \to \mathsf{tx}_{\mathsf{lock}}$. $\mathsf{CreateLockTx}$ is used to lock an account to an Ethereum address. It takes a secret key sk and an address addr as inputs. It outputs $\mathsf{tx}_{\mathsf{lock}} = (\mathsf{pkOf}(\mathsf{sk}), \mathsf{addr}, ...)$.
6. $\mathsf{CreateUnlockTx}(\mathsf{pk}) \to \mathsf{tx}_{\mathsf{unlock}}$. $\mathsf{CreateUnlockTx}$ is used to unlock an account. It takes a public key pk as input. It outputs $\mathsf{tx}_{\mathsf{unlock}} = (\mathsf{pk}, ...)$.

7. $\mathsf{ReadBalance}(\mathsf{sk},\mathsf{st}_h) \to b$. ReadBalance is used to find the balance of an account. It takes a secret key sk and state st_h as inputs, and outputs an integer b.

Zether Smart Contract. The smart contract has five functions Fund, Transfer, Burn, Lock and Unlock. They take $\mathsf{tx}_{\mathsf{fund}}$, $\mathsf{tx}_{\mathsf{trans}}$, $\mathsf{tx}_{\mathsf{burn}}$, $\mathsf{tx}_{\mathsf{lock}}$ and $\mathsf{tx}_{\mathsf{unlock}}$, respectively. These functions output 1 or 0, denoting success and failure respectively. If any of the inputs are not of the correct type the function automatically fails. Moreover the functions check certain properties of the input, such as verifying a proof or checking a nonce. If any of these checks fail, the function outputs 0. The five functions modify the state st as needed. We use SC as a shorthand for the smart contract.

SC has access to the current block height and the sender of every transaction. (In Solidity, the syntax for these are block.number and msg.sender, respectively.) It makes use of two constants: maximum amount value MAX and epoch length E, where $\mathsf{E} \geq 1$. The epoch number of a block at height h is defined to be $\lfloor h/\mathsf{E} \rfloor$. Thus, for example, the blocks at heights $0,1,...,\mathsf{E}-1$ are in the first epoch, the ones at heights $\mathsf{E},\mathsf{E}+1,...,2\mathsf{E}-1$ are in the second epoch, and so on.

We now discuss informally the correctness and security requirements of a payment mechanism. See the full version of this paper [Ano] for formal definitions. The definitions are for the more general case of anonymity, where not only the transfer amount but the sender/receiver are also hidden (among a chosen set of public keys).

Correctness. Correctness captures the basic functionality a payment mechanism should provide if transactions are generated honestly but they could be sent from arbitrary Ethereum addresses and processed in an arbitrary order. We will assume, however, that if a transaction is generated in a certain epoch, then it gets processed in the same epoch. To illustrate, suppose Alice has X ZTH in her account. In an epoch e_1, she publishes a transfer transaction to send $Y \leq X$ ZTH to someone else. There could be other users in the network who transfer to Alice at about the same time. Even if some of these transfers are processed before Alice's, we don't want her transfer to fail. Further, suppose Alice receives Z ZTH from others in e_1. Then, in any epoch after e_1, if she publishes a burn transaction with amount $X-Y+Z$, then we would like her to get back that amount of ETH.

To specify correctness formally, we define the notion of an *ideal state* and describe how it evolves over time as honestly generated transactions are processed. The ideal state tracks the balance of every account and the Ethereum address (if any) to which it is locked. When a transaction is processed, the ideal state is updated depending on the type of transaction and the current state. Informally, we say that a payment mechanism is correct if whenever a burn transaction is processed for a certain account, the amount of Ether returned to the user is equal to the amount of Zether held in the *ideal state account*.

Security Requirements. We define two security requirements for a payment mechanism Π, overdraft-safety and privacy. Overdraft-safety ensures that users cannot misuse the smart contract to withdraw more money from their accounts that they rightfully own. Privacy of a payment-mechanism ensures that no additional information about the payments of honest parties beyond the intended is leaked to an adversary. In other words, only the sender and receiver of a

payment should know the amount transferred in the payment. The formulation of overdraft-safety and privacy is inspired by Zerocash [BCG+14].

We define a game between a challenger Chal and an adversary Adv to capture the requirements, where Chal represents the honest users in the network. Both Chal and Adv have access to an oracle $\mathcal{O}_{\mathsf{SC}}$ who maintains the smart contract SC. Adv has full view of the oracle: it can see all the transactions sent by Chal to SC, how the state of SC changes, etc. We provide Adv substantial control over SC's state. It can instruct any honest party at any time (via the challenger) to publish a transaction. It can create its own malformed transactions based on the transactions of honest parties, and then push the former into the blockchain ahead of the latter. In particular, it can arbitrarily delay the transactions of honest parties.

For overdraft-safety, we associate some quantities with the game with respect to Adv: EtherDeposited, ZetherReceived and EtherBurnt, which have self-explanatory names. Informally, a payment mechanism is safe against overdrafts if

$$\mathsf{EtherDeposited} + \mathsf{ZetherReceived} \geq \mathsf{EtherBurnt}.$$

There are two important things to note here. First, it is not enough to just require that the *total* ether burnt (honest parties and adversary combined) should be no more than the *total* ether deposited because it could still be possible that the adversary is able to burn more than its fair share. Second, we cannot take the more direct approach of computing the amount adversary can withdraw by just reading the balance of the accounts controlled by it from the smart contract because its secret keys are not available.

We slightly modify the game discussed above to capture the privacy requirement. Instead of sending just one instruction to Chal every time (asking an honest party to create a transaction), Adv sends two *consistent* instructions at some point. Chal executes the $(b+1)$-th instruction based on a bit b hidden from Adv, which is chosen randomly in advance. Adv is supposed to guess b at the end of the game. (This is the typical left-or-right setting used for indistinguishability-based definitions.) Consistency is defined carefully to rule out trivial wins for the adversary.

D Security Proof

In this section, we show that Zether satisfies the correctness and security requirements from Appendix C.

Correctness. We first prove that Zether satisfies the correctness definition. Consider a slightly modified version of Zether where RollOver is called on all accounts at the end of each epoch. This only differs from Zether from an efficiency viewpoint as Zether implements lazy roll overs (i.e., every ZSC method rolls over all the accounts it touches in the very beginning of the call). Using this we show that every honestly generated transaction will in fact be processed successfully by ZSC.

Algorithms CreateTransferTx and CreateBurnTx roll over all the input public keys based on the state of the smart contract. Thus, any pending transfers associated with these keys are absorbed into the respective accounts and any pending lock requests take effect. Transfer and burn transactions are then generated

with respect to this new state of the accounts, which will match with the state ZSC will use to process them (as long as the delay is less than the length of an epoch).

Honest users put only those accounts in their anonymity set that are locked to the same address (if any of them is locked at all). Even if one of the account holders changes the lock on his/her account by calling Lock or Unlock, these methods treat the new locking address (which could be $\perp$) as a pending lock. The lock request will take effect in a subsequent epoch, so transactions generated in this epoch will not be affected.

The rest of the correctness follows from the homomorphic properties of ElGamal encryption as well as the correctness properties of the proof system. Though the encrypted values are in $\mathbb{Z}_p$ and the ideal state handles positive integers, this is not a problem because ZSC takes deposits only up to an amount MAX, a constant much smaller than p. The homomorphic operations, therefore, would not cause an overflow.

Finally, note that a user is able to create a nonce and as such a transaction or burn per epoch unless $g_{\mathsf{epoch}} = g_{\mathsf{epoch}'}$ for $\mathsf{epoch} \neq \mathsf{epoch}'$. This however happens with at most negligible probability if the hash function $\mathcal{H}$ is collision resistant.

Overdraft Safety. We show that ZSC methods move the right amount of funds to/from accounts by proving that they satisfy certain properties. An inductive argument would then show that Zether is safe against overdrafts.

Let us consider the method Fund first. Let (C_L,C_R) be the (rolled over) state of an account y. If, hypothetically, Burn is invoked on this state, suppose it returns b ETH. Now Fund is called with b' ETH. We show that if Burn is invoked again (hypothetically), it will return no more than $b+b'$ ETH. Since Burn returns b on the first invocation, it must be that $C_L = g^b C_R^{\mathsf{sk}}$ due to the soundness property of ZK-proofs. When b' is deposited, pTransfers is set to $(g^{b'},1)$. Now when Burn is invoked again, the state of y will either be (C_L,C_R) or $(C_L,C_R)\circ(g^{b'},1)$ depending on whether there is a roll over or not. In the first case, only b will satisfy the required relation between C_L and C_R, and, in the second case, only $b+b'$ will. So, again due to the soundness property, at most $b+b'$ can be obtained by burning.

Next, we consider the method Transfer. Let $(y_1,...,y_n)$ be the anonymity set, (C_1,D), ..., (C_n,D) be the ciphertexts, and π_{transfer} be the proof for a transfer transaction tx. Let $(C_{L,i},C_{R,i})$ be the (rolled over) state of the concerned accounts. If Burn is invoked (hypothetically) on these accounts, suppose it returns $b_1,...,b_n$ ETH, respectively. Now, if tx is processed successfully by Transfer, then it must be that there exists a j, k and $b^\star$ s.t. (C_j,D) encrypts $-b^\star$ under y_j, (C_k,D) encrypts $b^\star$ under y_k, and rest of the ciphertexts encrypt 0 (due to the soundness property). Transfer sets $\mathsf{pTransfers}[y_i]$ to be (C_i,D) for all i.

Thus, when Burn is invoked again on y_i, its state will either be $(C_{L,i},C_{R,i})$ or $(C_{L,i},C_{R,i})\circ(C_i,D)$ depending on whether there is a roll over or not. For the accounts other than y_j and y_k, the same amount as before will be returned. For y_k, at most $b_k+b^\star$ will be returned. Finally, for y_j, note that no burning can take place in this epoch because transfer has already declared the nonce. When a burn is processed in the next epoch, there will be a roll over changing the account state to $(C_{L,i},C_{R,i})\circ(C_i,D)$. So Burn will return $b_j - b^\star$. Therefore, we can see that transfer

transactions cannot be used to increase the overall Zether balance of the accounts involved. Further note that the nonce along with the soundness of the proof system, enforce that an adversary will at most be able to do a single transfer per account per epoch. One can similarly analyze the method Burn. We skip the details.

We can now use a simple inductive argument to show that an adversary which wins Security-Game will break the soundness of the proof system. This happens with at most negligible probability.

Privacy: Confidentiality & Anonymity. In Privacy-Game, Adv sends one instruction to Chal every time except once, when it sends two consistent instructions. The consistency requirements prevent Adv from trivially winning the game. If the instructions are for funding, locking or unlocking, then it is easy to see that the adversary has no advantage. Two consistent burn instructions will also not reveal any additional information to Adv due to the zero-knowledge property of the proof system.

We are only left with the case of two consistent transfer instructions. A transfer transaction consists of an anonymity set $\mathbf{y}$, a list of commitments $\mathbf{C}$, a blinding value D, a nonce u, and π_{transfer}. Two consistent transactions could have two different senders, so the nonce values could be different. However, g_{epoch}^{x} (for any x) is indistinguishable from random under the DDH assumption since both y and g_{epoch} are random (when $\mathcal{H}$ is modeled as a random oracle). Further, ciphertexts (C_i,D) for honest i are indistinguishable from the encryption of random messages. Now, let the receivers for the two instructions be j and k. If neither of them are under the control of Adv, then all the ciphertexts Adv can decrypt are just encryptions of 0. Otherwise, both j and k must be corrupt. In this case, Adv can decrypt (C_j,D) and (C_k,D) too, but then they must decrypt to the same amount.

E Σ-Bullets

Bulletproofs [BBB+18] enable proofs on Pedersen committed values by computing a linear combination of commitments and opening that combination. This uses the homomorphic property of Pedersen commitments that use the same commitment key. The core idea of Σ-Bullets is to replace this linear combination with a Σ-protocol. The Σ-protocol ensures that the linear combination of the encoded values is equal to some public value. The efficient composability of Σ-protocols allows us to combine the opening with other proofs.

We first present a high-level overview of the Bulletproofs protocol and then discuss how we modify it to construct Σ-Bullets. The prover first commits to the circuit's wires in A and to a vector of blinding values in S. The commitments are Pedersen vector commitments [BCC+16]. The prover then receives challenges y,z and commits to polynomial $t(X)$ using a polynomial commitment that can be verifiably opened to an evaluation of $t(X)$. The prover does not commit to one of the coefficients of the polynomial, e.g. the 0 coefficient. If the prover is honest then the verifier can compute said coefficient from just the challenges and the commitments to wire values v_is which are committed to in V_i.

Finally, the prover convinces the verifier that $t(X)$ is equal to the inner product of two polynomials with vector coefficients. The two polynomials can

be homomorphically constructed from A, S and the challenges. This final step uses an inner product argument which requires only $O(\log(n))$ communication where n is the size of the circuit. The protocol can be made non-interactive using the Fiat-Shamir heuristic. Given an arithmetic circuit $\mathsf{Circuit}:\mathbb{Z}_p^n \times \mathbb{Z}_p^m \to \{0,1\}$, the prover wants to prove the following that she knows $\boldsymbol{a} \in \mathbb{Z}_p^n, \boldsymbol{v} \in \mathbb{Z}_p^m$ such that

$$V_i = \mathsf{Encode}(v_i) \forall i \in [1,m], \mathsf{Circuit}(\boldsymbol{a},\boldsymbol{v}) = 1.$$

Now, the verifier creates commitment P to $l(X)$,$r(X)$ from A,S,y,z and checks the following condition:

1. $T(0) = t_0$
2. $Open(T \cdot \prod_{i=1}^{m} V_i^{z^i}) + \delta(y,z) = \hat{t}$
3. $Open(P) = \boldsymbol{l},\boldsymbol{r}$
4. $\langle \boldsymbol{l},\boldsymbol{r} \rangle = \hat{t}$

Note that the second condition requires that T and the V_i's are additively homomorphic. We can therefore not simply replace T and V_i with ElGamal encryptions as they are not homomorphic if done under different keys. We generalize the protocol by simply requiring that the prover *proves* that

$$\hat{t} = \sum_{i=1}^{m} v_i \cdot z^i + \delta(y,z) + Open(T). \tag{3}$$

Instead of giving the opening of the polynomial commitment to $t(X)$, the verifier instead proofs knowledge of the opening. Concretely, this equates to opening the Pedersen commitment $T = g^t h^\tau$ by proving knowledge of the blinding value τ. This can be achieved through a simple Σ-protocol. We show concretely how this can be achieved for $m = 2$ and V_1,V_2 being ElGamal encryptions in the burn proof. We further provide a security proof for that protocol. The security proof for the overall protocol is straightforward if the Σ-protocol proves a statement that implies (3) while having special-soundness and zero-knowledge properties. The Bulletproof extractor simply extracts the openings of the V's from the Σ-protocol and otherwise proceeds as described in [BBB+18]. The simulator uses the Σ-protocol's simulator to generate a valid looking Σ-protocol as a sub-routine.

We now proceed by presenting two Σ-protocols that are vital for Zether.

Proof Sketch. We will show that the sigma protocol is perfectly complete, honest-verifier zero-knowledge and has special-soundness. Completeness is immediate. For zero-knowledge we build a simulator S that constructs valid and indistinguishable transcripts given a valid statement $(y, C_L, C_R, u, b, g, g_{\mathsf{epoch}})$ and access to the verifier's state. The simulator will first samples a random challenge c a random s_{sk} and computes $A_y = g^{s_{\mathsf{sk}}} y^{-c}$,$A_{C_R} = C_R^{s_{\mathsf{sk}}} (\frac{C_L}{g^b})^{-c}$ and $A_u = g_{\mathsf{epoch}}^{s_{\mathsf{sk}}} u^{-c}$. If the verifier is honest, i.e. generates random challenges then c,s_{sk} are uniformly distributed and A_y,A_{C_R},A_u form DDH tuples. The simulated transcripts are identically distributed. To prove special-soundness we build an extractor that can compute the witness from two accepting transcripts with the same first round message A_y, A_{C_R}, A_u. The transcripts also contain (c, s_{sk}) and (c', s'_{sk}) respectively. If both transcripts are accepting then the extractor can output $\hat{\mathsf{sk}} = \frac{s_{\mathsf{sk}} - s'_{\mathsf{sk}}}{c - c'}$ as a valid witness. We can directly deduce from the verification equations that $g^{\hat{\mathsf{sk}}} = y \wedge C_R^{\mathsf{sk}} = \frac{C_L}{g^b} \wedge g_{\mathsf{epoch}}^{\hat{\mathsf{sk}}} = u$.

Σ-Bullets ConfTransfer proof. We will now describe how exactly we use the Σ Bullets construction to create an efficient proof for $\mathsf{st}_{\mathsf{Transfer}}$ the statement which proofs that a ConfTransfer transaction is valid. We will use Bulletproofs to perform the range proofs and use a sigma protocol to proof that the balances are properly encrypted. The conjunction of these two is $\mathsf{st}_{\mathsf{ConfTransfer}}$.

The Σ-protocol takes as input the senders public key y the receiver's public key $\hat{y}$ and an encryption of the senders balance *after* the transfer $C_{L,n} = \frac{C_L}{C}, C_{R,n} = \frac{C_R}{D}$. Further it takes the encryption of the in and outgoing amounts as input, i.e. $C, D, \overline{C}$. Then the bulletproof protocol is run as described above. The Σ then also takes in T the commitment to $t(X)$ as well as an opening of it at the challenge x: $(\hat{t})$. Note that it is important that the Σ-protocol is run after the Bulletproofs protocol. In the non-interactive variant this means that the whole Bulletproofs transcript is also hashed in order to generate the Σ-protocol challenge.

$$\Big\{(y,\overline{y},C_{L,n},C_{R,n},C,D,\overline{C},z,\hat{t},\delta(y,z);\mathsf{sk},b^\star,b',r,\tau):$$
$$C = g^{b^\star} y^r \wedge \overline{C} = g^{b^\star}\overline{y}^r \wedge D = g^r \wedge$$
$$C_{L,n} = g^{b'} C_{R,n}^{\mathsf{sk}} \wedge g^{\mathsf{sk}} = y \wedge$$
$$t = \hat{t} - \delta(y,z) \wedge g^{t - b^\star \cdot z^2 - b' \cdot z^3} h^\tau = T_{1,2}\Big\} \quad (4)$$

Prover$(\mathsf{sk}, r, b^\star, b')$		**Verifier**
$k_{\mathsf{sk}}, k_r, k_b, k_\tau \leftarrow\$ \quad \mathbb{Z}_q$		
$A_y = g^{k_{\mathsf{sk}}}$		
$A_D = D^{k_r}$		
$A_b = g^{k_b} D^{-k_{\mathsf{sk}} z^2} C_{R,n}^{-k_{\mathsf{sk}} z^3}$		
$A_{\bar{y}} = (\frac{y}{\bar{y}})^{k_r}$		
$A_t = g^{-k_b} h^{k_\tau}$		
	A_y, A_D $A_b, A_{\bar{y}}, A_t$ $\longrightarrow$	
	$\longleftarrow$ c	$c \leftarrow\$ \quad \mathbb{Z}_q$
$s_{\mathsf{sk}} = k_{\mathsf{sk}} + c\mathsf{sk}$		
$s_r = k_r + cr$		
$s_b = k_b + c(b^\star z^2 + b' z^3)$		
$s_\tau = k_\tau + c \cdot \tau$		
	$\longleftarrow$ $s_{\mathsf{sk}}, s_r, s_b, s_\tau$	
		Check the following:
		$g^{s_{\mathsf{sk}}} = A_y y^c$
		$g^{s_r} = A_D D^c$
		$g^{s_b} (D^{z^2} C_{R,n}^{z^3})^{s_{\mathsf{sk}}} = A_b (C^{z^2} C_{L,n}^{z^3})^c$
		$(\frac{y}{\bar{y}})^{s_r} = A_{\bar{y}} (\frac{C}{\overline{C}})^c$
		$g^{t \cdot c - s_b} h^{s_\tau} = A_t T_{1,2}^c$

Proof Sketch. We now provide a sketch of the proof that our Σ-protocol is secure and is a proof of knowledge for Relation (4). The protocol is honest verifier zero knowledge because there exists a simulator that can simulate verifying transcripts without access to the witness. The simulator generates a random challenge c and random $s_{\mathsf{sk}}, s_r, s_b, s_\tau$. He then computes $A_y, A_D, A_b, A_{\hat{y}}$ and A_t according to the verification equations. If $g, D, \frac{y}{\hat{y}}$ and h are group generators, i.e. not equal to 1, then each A value is a random group element in the honest protocol and in the simulated transcript. A_D and $A_{\hat{y}}$ form a DDH tuple with basis g and $\frac{y}{\hat{y}}$ which means that they are computationally indistinguishable from the independently sampled A_D an $A_{\hat{y}}$.

We prove that the protocol is a proof of knowledge by showing that we can build an extractor. The extractor rewinds the sigma protocol once to receive two accepting transcripts with different challenges and the same first message. Let $c_2, s_{\mathsf{sk},2}, s_{r,2}, s_{b,2}, s_{\tau,2}$ be the second transcript. From them he computes $\mathsf{sk} = \frac{s_{\mathsf{sk}} - s_{\mathsf{sk},2}}{c - c_2}, r = \frac{s_r - s_{r,2}}{c - c_2}, b = \frac{s_b - s_{b,2}}{c - c_2}, \tau = \frac{s_\tau - s_{\tau,2}}{c - c_2}$. From the verification equations we can deduce that $y = g^{\mathsf{sk}}, D^r, g^b = (\frac{C}{D^{\mathsf{sk}}})^{z^2} (\frac{C_{L,n}}{C_{R,n}^{\mathsf{sk}}})^{z^3}$ Further we also have that $g^{\hat{t}} h^\tau = g^{b + \delta(y,z)} T_{1,2}$. In order to extract $b^\star$ and b' we need to rewind the whole Σ-Bullets protocol twice to get three executions with different zs: (z_1, z_2, z_3). Using the same extraction procedure for the Σ-protocol we get the extracted witnesses $(\mathsf{sk}_i, r_i, b_i, \tau_i), i \in \{1,2\}$. First note that $\mathsf{sk}_1 = \mathsf{sk}_2$ since $g^{\mathsf{sk}_1} = g^{\mathsf{sk}_2} = y$. We can now form the equations:

$$g^{b_1} = (\frac{C}{D^{\mathsf{sk}}})^{z_1^2} (\frac{C_{L,n}}{C_{R,n}^{\mathsf{sk}}})^{z_1^3}, \quad g^{b_2} = (\frac{C}{D^{\mathsf{sk}}})^{z_2^2} (\frac{C_{L,n}}{C_{R,n}^{\mathsf{sk}}})^{z_2^3}.$$

One can now easily find a linear combination of these equations to compute $b^\star$ such that $g^{b^\star} = \frac{C}{D^{\mathsf{sk}}}$ and b' such that $g^{b'} = \frac{C_{L,n}}{C_{R,n}^{\mathsf{sk}}}$. This shows that we can successfully extract a witness $(\mathsf{sk}, b^\star, b', r)$ such that the statement $(y, \overline{y}, C_{L,n}, C_{R,n}, C, D, \overline{C}, z, \hat{t}, \delta(y,z); \mathsf{sk}, b^\star, b', r, \tau)$ is in Relation (4).

E.1 Signatures

Zether not only uses zero-knowledge proofs but also heavily relies on signatures. Instead of instantiating a separate signature scheme, we can leverage our zero-knowledge proofs to also provide signature functionality. All of the ZK-proofs in our instantiation of Zether are derived from interactive proofs and then transformed to non-interactive proofs using the Fiat-Shamir heuristic [FS87, BCS16]. The Fiat-Shamir heuristic and its extension to multi-round protocol transform an interactive public-coin proof into a non-interactive proof by generating the verifiers' messages from the hash of the transcript. There exists a simple transformation that creates a signature scheme from such a proof system [AABN02]. Concretely, if the prover shows knowledge of a private key and then appends the message to the transcript before generating the challenge, then the proof also acts as a signature. This leads to signatures that can be generated and verified at almost no additional cost.

F Other Remarks

Rolling Over Pending Transfers. We define a separate (internal) method for rolling over, and the first thing every other method does is to call this method.

There could be accounts that do not get rolled over for several consecutive epochs because no transaction is initiated from them. This is not a problem because the account holder, say Alice, is not trying to use her money anyway. At some later point in time, when Alice wants to operate on her account, she will publish a transaction. All the money transferred into her account since the last rollover will be rolled over immediately and become available to be spent. Indeed, when Alice creates a ZK-proof, she will assume the state of her account to be what it would be when all the pending transfers are rolled over into it.

Note on Replay Protection. One way wonder if there is a way to use Ethereum addresses themselves as the identities of Zether accounts. The accounts would then be operated with the secret keys corresponding to the addresses, and we would get replay protection and signature verification for free. However, this would force users to operate a Zether account from a fixed Ethereum address. They would not be able to delegate the account to a different address, like when locking the account to a smart contract (which will be discussed in more detail later). Furthermore, Ethereum addresses are only a hash of public keys, not the full form. Proving statements about hashes in zero-knowledge is quite expensive. Lastly, having separate public keys for Zether accounts also helps make the design more modular and platform-independent.

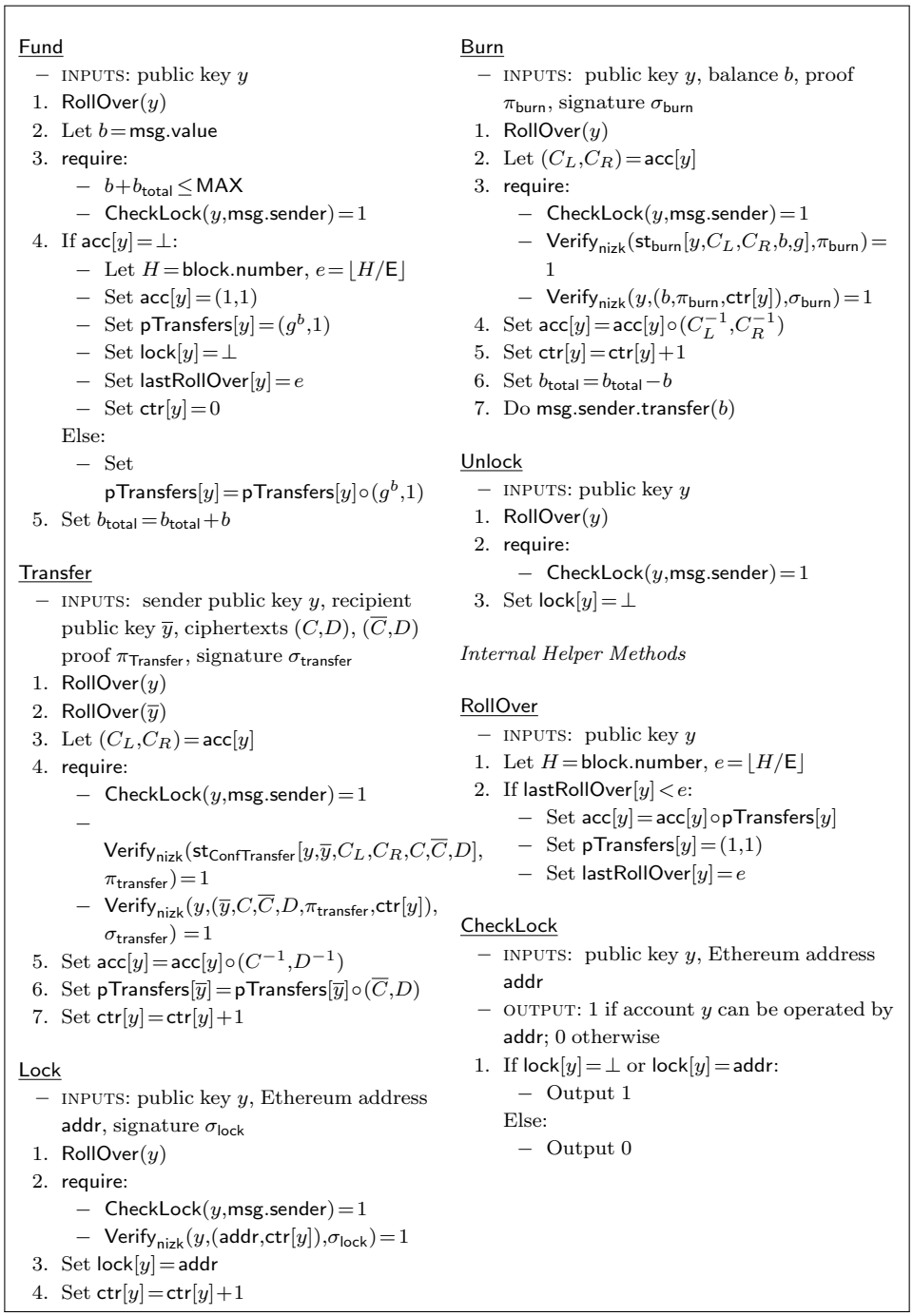
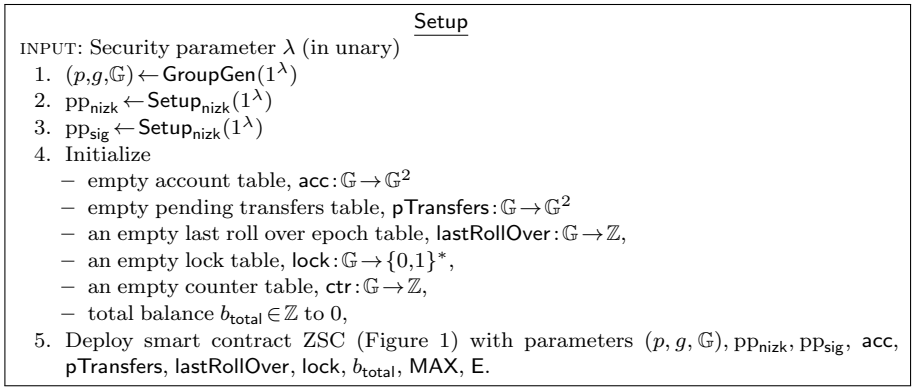


Fig. 1. ZSC: The Zether smart contract

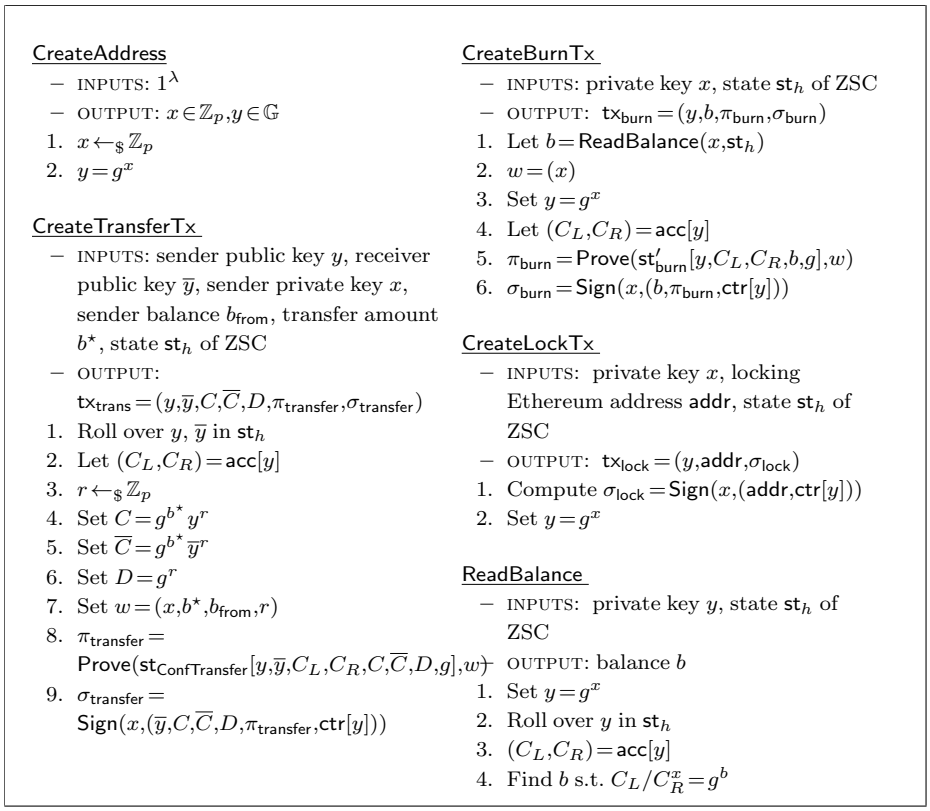


Setup

INPUT: Security parameter λ (in unary)

1. $(p,g,\mathbb{G}) \leftarrow \mathsf{GroupGen}(1^\lambda)$
2. $\mathsf{pp}_{\mathsf{nizk}} \leftarrow \mathsf{Setup}_{\mathsf{nizk}}(1^\lambda)$
3. $\mathsf{pp}_{\mathsf{sig}} \leftarrow \mathsf{Setup}_{\mathsf{nizk}}(1^\lambda)$
4. Initialize
   - empty account table, $\mathsf{acc}: \mathbb{G} \to \mathbb{G}^2$
   - empty pending transfers table, $\mathsf{pTransfers}: \mathbb{G} \to \mathbb{G}^2$
   - an empty last roll over epoch table, $\mathsf{lastRollOver}: \mathbb{G} \to \mathbb{Z}$,
   - an empty lock table, $\mathsf{lock}: \mathbb{G} \to \{0,1\}^*$,
   - an empty counter table, $\mathsf{ctr}: \mathbb{G} \to \mathbb{Z}$,
   - total balance $b_{\mathsf{total}} \in \mathbb{Z}$ to 0,
5. Deploy smart contract ZSC (Figure 1) with parameters $(p, g, \mathbb{G})$, $\mathsf{pp}_{\mathsf{nizk}}$, $\mathsf{pp}_{\mathsf{sig}}$, acc, $\mathsf{pTransfers}$, $\mathsf{lastRollOver}$, lock, b_{total}, MAX, E.

Fig. 2. Zether setup



CreateAddress

- INPUTS: 1^λ
- OUTPUT: $x \in \mathbb{Z}_p, y \in \mathbb{G}$

1. $x \leftarrow_\$ \mathbb{Z}_p$
2. $y = g^x$

CreateTransferTx

- INPUTS: sender public key y, receiver public key $\overline{y}$, sender private key x, sender balance b_{from}, transfer amount $b^\star$, state st_h of ZSC
- OUTPUT: $\mathsf{tx}_{\mathsf{trans}} = (y,\overline{y},C,\overline{C},D,\pi_{\mathsf{transfer}},\sigma_{\mathsf{transfer}})$

1. Roll over y, $\overline{y}$ in st_h
2. Let $(C_L,C_R) = \mathsf{acc}[y]$
3. $r \leftarrow_\$ \mathbb{Z}_p$
4. Set $C = g^{b^\star} y^r$
5. Set $\overline{C} = g^{b^\star} \overline{y}^r$
6. Set $D = g^r$
7. Set $w = (x,b^\star,b_{\mathsf{from}},r)$
8. $\pi_{\mathsf{transfer}} = \mathsf{Prove}(\mathsf{st}_{\mathsf{ConfTransfer}}[y,\overline{y},C_L,C_R,C,\overline{C},D,g],w)$
9. $\sigma_{\mathsf{transfer}} = \mathsf{Sign}(x,(\overline{y},C,\overline{C},D,\pi_{\mathsf{transfer}},\mathsf{ctr}[y]))$

CreateBurnTx

- INPUTS: private key x, state st_h of ZSC
- OUTPUT: $\mathsf{tx}_{\mathsf{burn}} = (y,b,\pi_{\mathsf{burn}},\sigma_{\mathsf{burn}})$

1. Let $b = \mathsf{ReadBalance}(x,\mathsf{st}_h)$
2. $w = (x)$
3. Set $y = g^x$
4. Let $(C_L,C_R) = \mathsf{acc}[y]$
5. $\pi_{\mathsf{burn}} = \mathsf{Prove}(\mathsf{st}'_{\mathsf{burn}}[y,C_L,C_R,b,g],w)$
6. $\sigma_{\mathsf{burn}} = \mathsf{Sign}(x,(b,\pi_{\mathsf{burn}},\mathsf{ctr}[y]))$

CreateLockTx

- INPUTS: private key x, locking Ethereum address addr, state st_h of ZSC
- OUTPUT: $\mathsf{tx}_{\mathsf{lock}} = (y,\mathsf{addr},\sigma_{\mathsf{lock}})$

1. Compute $\sigma_{\mathsf{lock}} = \mathsf{Sign}(x,(\mathsf{addr},\mathsf{ctr}[y]))$
2. Set $y = g^x$

ReadBalance

- INPUTS: private key y, state st_h of ZSC
- OUTPUT: balance b

1. Set $y = g^x$
2. Roll over y in st_h
3. $(C_L,C_R) = \mathsf{acc}[y]$
4. Find b s.t. $C_L / C_R^x = g^b$

Fig. 3. User algorithms of Zether

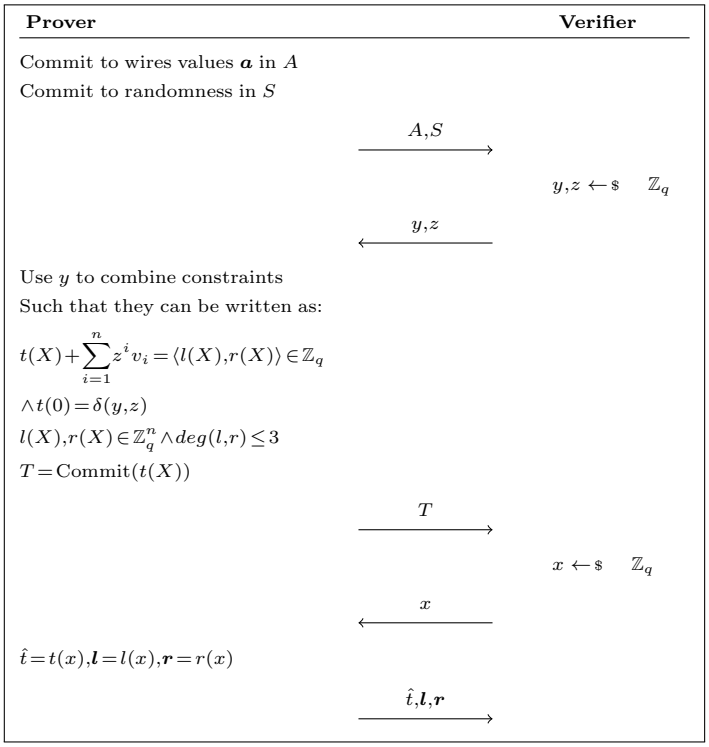


Fig. 4.

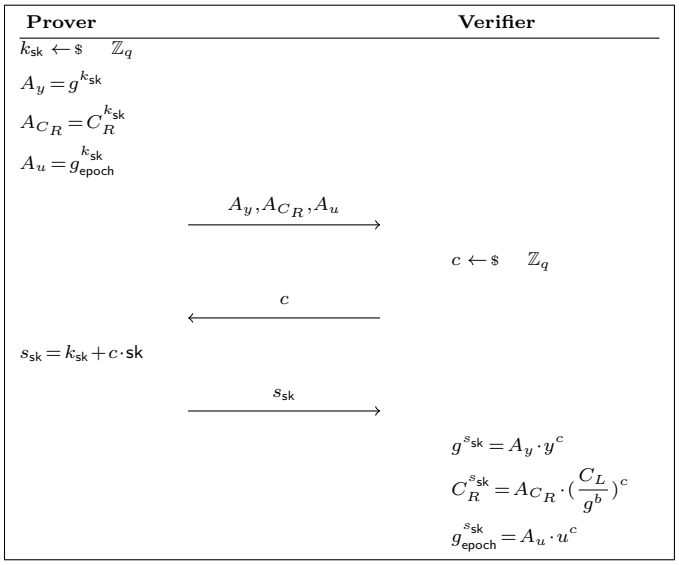


Fig. 5. Σ-protocol for proving $\mathsf{st}_{\mathsf{Burn}}$ statement