Zether: Towards Privacy in a Smart Contract World

Benedikt Bünz[1], Shashank Agrawal[2], Mahdi Zamani[3], and Dan Boneh[4]

[1]Stanford University, benedikt@cs.stanford.edu
[2]Visa Research, shaagraw@visa.com
[3]Visa Research, mzamani@visa.com
[4]Stanford University, dabo@cs.stanford.edu

Abstract

Decentralized smart contract platforms such as Ethereum have recently become immensely popular. Unlike Bitcoin's transaction-based system, smart contracts are arbitrary pieces of code that sit on a blockchain and facilitate fine-grained control over the distribution of assets. Unfortunately, as of today, there is no fully-decentralized way to conduct transactions privately on such a powerful platform.

In this paper, we propose *Zether*, a private payment mechanism that is compatible with Ethereum and other account-based payment systems. Zether can provide both confidentiality (by hiding payment amounts) and anonymity (by hiding the identities of senders and recipients). We take an account-based approach similar to Ethereum for efficiency and usability. Namely, we implement a new Ethereum smart contract that stores confidential account information for users and provides privacy-preserving functionalities for depositing, transferring, and withdrawing funds to/from these accounts with only a small overhead.

We describe techniques to protect Zether against replay attacks and front-running situations and provide mechanisms to allow Zether inter-operate with arbitrary smart contracts to support applications such as sealed-bid auctions, payment channels, stake voting, and confidential proof-of-stake. As a part of our protocol, we propose Σ-Bullets, an improvement of the existing zero-knowledge proof system, Bulletproofs. Σ-Bullets make Bulletproofs more inter-operable with Sigma protocols, which is of general interest.

We implement Zether as an Ethereum smart contract and show the practicality of our design by measuring the amount of *gas* used by the Zether contract. A Zether confidential transaction currently costs about 0.0129 ETH or approximately \$2.72. We discuss how small changes to Ethereum, which are already being discussed independently of Zether, would drastically reduce this cost.

1 Introduction

The demand for cryptocurrencies has increased dramatically in recent times. The total market capitalization of all cryptocurrencies is at roughly 227B USD as of late 2018, up from 18B USD in early 2017 [50]. These cryptocurrencies use a distributed ledger, known as a *blockchain*, to record all transactions, which can be verified and extended by any entity, making them open and transparent. The same openness, however, comes with a significant privacy cost. In the two most popular cryptocurrencies, Bitcoin [59] and Ethereum [17], for instance, not only are the transaction amounts fully visible in the clear but users can also be de-anonymized [22, 31, 55, 71]

While both suffer from privacy issues, Bitcoin and Ethereum have a very different design. Nakamoto's primary motivation was to build a decentralized value-transfer mechanism that enables payments without a trusted third party [59]. While Bitcoin is a special type of transaction-based state machines, Ethereum attempts to build the "generalized" mechanism on top of which any transaction-based state machines may be built [83]. More concretely, Ethereum natively allows users to build *smart contracts*, computer programs that directly control digital assets [79]. Smart contracts allow running applications such as auctions, lotteries, and crowd-sales [32, 34], while doing the same on Bitcoin is quite complicated and inefficient [57], if not impossible.

Ethereum's state is a mapping between addresses and accounts that store account-specific information such as balances and program states. When an Ethereum user, Alice, creates a transaction to transfer some ether, her transaction is checked against the mapping to verify that her account has enough balance. This is usually referred to as the *account-based* model as opposed to the *unspent transaction output (UTXO)* model of Bitcoin, where every new transaction spends one or more UTXOs, all of which are stored on the blockchain. To spend five Bitcoins, user Bob must use past transactions addressed to him whose unspent portions sum to at least five. On account of being stateless, UTXOs are not well-suited for applications that are more complex than asset issuance and transfer such as various kinds of smart contracts, which are generally stateful [19]. The account-based model is taken by several other currencies and our techniques apply to them as well.

Coming back to the privacy question, almost all the privacy-preserving mechanisms developed so far in the context of blockchains are in the Bitcoin model [3, 12, 42, 43, 51, 52, 60, 64, 73, 82]. These mechanisms can hide payment amounts (i.e., provide confidentiality) and/or the sender/receiver (i.e., provide anonymity). They require major changes to Bitcoin's design and have often spun off into separate cryptocurrencies. With Ethereum on the other hand, one could hope to build a privacy mechanism in the form of a smart contract, which would then run on Ethereum like any other application.

Alternatively, one may consider implementing one of the existing privacy-preserving mechanisms developed in the UTXO model as an Ethereum smart contract to hide the payment amounts and perhaps even the sender/receiver. While such a strategy may help us save the effort involved in designing new mechanisms, there are several key advantages in designing a privacy-preserving mechanism in the account-based model rather than the UTXO model:

- *Storage.* Storage on the Ethereum blockchain is quite expensive (for good reasons) and should be minimized as much as possible. For example, storing only 256 bits of information costs roughly 20,000 gas units (see Section 2 for the definition of gas), whereas sending it costs only 2,176 gas units [83]. If a UTXO-based model is implemented, then all unspent transactions need to be stored on a smart contract. Moreover, for all UTXO-based currencies that provide cryptographic anonymity [3, 60], all *spent* transactions have to be stored as well. This makes the smart contract expensive to use.

- *Confidentiality.* Accounts provide a natural *mixing* property. A new transaction in the account-based model draws from the total available balance, whereas the UTXOs used to create a new transaction in the Bitcoin model could reveal a lot of information about the new transaction, even if all the amounts are cryptographically hidden. For example, if a UTXO (even encrypted) coming from a user's employer is used to create a new transaction, then the user is likely going to spend a significant amount of money. On the other hand, though UTXOs could potentially provide better anonymity, users can often

be de-anonymized in practice [22, 31, 55, 71].

- *Interoperability with smart contracts.* An account-based privacy-preserving mechanism can easily inter-operate with smart contracts, which typically maintain accounts for their users and for the contract itself. For example, a private account can be used to fund an account on the smart contract, or be tied to a payment channel smart contract [5, 69]. Funds can be added to the channel by simply adding funds to the private account.

- *Adoption.* Four out of the six most valuable cryptocurrencies (as of late 2018) use an account-based model (Ethereum, Ripple [70], EOS [33], Stellar [53]). An account-based privacy mechanism can be used to introduce private payments to these platforms.

- *Simplicity for users.* If we provide privacy in the account-based model, then it encourages users to have a small number of accounts, making account management simpler. This also makes the system more efficient overall.

Privacy concerns in a smart contract platform such as Ethereum go beyond payments. While hiding payment details is certainly an important task, one may want to hide the state of a smart contract, how it changes over time, or even the functionality of the contract itself. This stronger kind of privacy is provided by some recent proposals like Hawk [46] and Ekiden [24], but they are not fully decentralized. Hawk relies on a trusted setup phase to generate a common random string and a trusted *manager* who gets access to the private inputs of parties. Ekiden relies on trusted execution environments such as Intel SGX at the core of its design. Zexe [15] is a recent proposal for a private scripting language for Zerocash [3] 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.

Indeed, achieving a high level of privacy for smart contracts is a challenging task, and a general-purpose mechanism that works for arbitrary contracts can be too expensive for simple ones. Thus, one may wonder if there is a simple, efficient, and fully-decentralized way to obtain a reasonable level of privacy for contracts that are commonly used. For example, people should be able to bid through an auction contract without revealing their bid values, at least in the bidding phase. They should be able to setup payment channels without revealing the amount of money in the channel.

Our Contribution. This paper makes the following contributions:

- We introduce *Zether*, a privacy-preserving payment mechanism that requires no changes to the design of the underlying smart contract platform (e.g., Ethereum).

- We describe Zether as a smart contract that can be executed either individually or by other smart contracts to exchange confidential amounts of a token, denoted by ZTH. Zether provides both confidentiality by hiding transaction amounts and anonymity by hiding the identity of the senders and the recipients. The techniques used in Zether can apply to all account-based cryptocurrencies, completely independent of their consensus mechanisms.

- To evaluate the practicality of Zether, we implement it 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. As part of our implementation we provide an efficient implementation of Bulletproofs [16] on Ethereum which is of independent interest.

- We propose a new zero-knowledge (ZK) proof mechanism, called Σ-Bullets, which enhances the interoperability of Σ-protocols and Bulletproofs [16]. This allows us to efficiently combine Bulletproofs-based range proofs with ElGamal encryptions and ring-signature based anonymous transfers.

- We use Zether to build four privacy-preserving applications: a decentralized auction mechanism that fully hides the bids (no collateral required), a confidential payment channel mechanism, a confidential stake voting protocol, and a private proof-of-stake protocol.

In this paper, we mainly focus on providing private payments for Ethereum-style smart contract platforms in the account-based model. As we explain in Section 3, similar techniques can be used to allow private payments in other account-based systems.

Organization. The paper is organized as follows. We provide some background on Ethereum in Section 2, with a focus on smart contract execution. Section 3 provides an overview of Zether: we describe the challenges we faced in building Zether and how we overcame them. Section 4 defines the cryptographic tools we need for Zether. It then formally defines the notion of a payment mechanism in a smart-contract world, and the security properties it should satisfy. Section 6 provides a detailed description of Zether. In Section 7, we describe a basic version of Zether, followed by a discussion on important matters related to the use of Zether in Section 8. Section 9 describes the Σ-bullets proof system. Section 10 describes our implementation of Zether in Solidity, its evaluation, various optimizations, etc. Finally, Section 11 shows how Zether can be used with smart contracts to get more privacy. We also provide formal security definitions and a security proof in the Appendix.

2 Background on Ethereum

Ethereum's basic units are the accounts. 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 [17,83]. 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 [17].

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 [83]. 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 [83].

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) [83].

Contracts are written in specific programming languages such as Solidity [77]. Once compiled to bytecode, the contract can be read and executed by the *Ethereum virtual machine (EVM)*, a sandboxed and isolated runtime 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 [49]. The low computational power, along with the asynchronous transactional nature of the Ethereum network make programming complicated smart contracts a delicate endeavor.

3 Overview of Zether

We now give a high-level overview of Zether, highlighting our design choices. Several proposals for making transactions confidential use homomorphic commitments, such as Pedersen commitments [62]. Though Pedersen 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. We take a more direct, and simpler, approach using ElGamal encryption with messages in the exponent [26]. The encryption scheme has linear encoding properties, i.e., it is homomorphic, which we utilize to create efficient ZK-proofs of correct encryption. We describe the encryption scheme in Section 4.

3.1 Transferring Confidentially

We first discuss a simplistic version of Zether with the core functionalities. We then describe some shortcomings and how to overcome them. Besides privacy, an important goal of Zether is, of course, to make sure that honest users never lose money.

A simplistic Zether . The Zether smart contract (ZSC) works with Zether tokens (ZTH). Zether accounts are identified with ElGamal public keys. To fund an account with public

key y with b ZTH, one can send b ETH to the smart contract. ZSC generates an ElGamal encryption of b with randomness 0 (since b is anyway part of the transaction) and *adds* it to the encrypted balance associated with y.[1] One can convert ZTH back to ETH by revealing the current balance $b^\star$ and providing a ZK-proof that y's ciphertext (i.e., the ciphertext associated with y on the smart contract) 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, they encrypt the same positive value, and the remaining balance associated with y is positive. In our implementation, we use a new ZK-proof system, called Σ-Bullets to efficiently prove the statements over the encrypted transfer balance and the new sender balance.

Incoming transfers. The very first problem with the simplistic version of Zether stems from the fact that the ZK-proofs are generated w.r.t. 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 w.r.t. to her current account balance, stored in an encrypted form on the contract. However, 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. Note that Bob may be a totally benign user 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: 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.

Any kind of locking approach becomes more untenable when we bring in anonymity. (We will discuss anonymity in more detail shortly.) If Alice wants to hide herself among other users and make sure that her transaction gets through, she will have to lock *all* the accounts in the anonymity set. Clearly, this cannot be allowed: Alice must not be able to lock accounts of other users. Alternatively, Alice could only put locked accounts in her anonymity set. However, if someone unlocks their account before Alice's transaction gets in, Alice's degree of anonymity is reduced.

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.

We introduce another idea here called *epochs*. We divide time into epochs where an epoch consists of k consecutive blocks. The choice of k depends on two factors: a) the gap between the latest state of blockchain and any user's view, and b) the time it takes to get a transaction into the blockchain. At the end of every epoch, pending transfers are rolled over into the

---

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

corresponding accounts. Users are expected to publish their transfer or burn transaction at the beginning of an epoch so that even if they do not see the latest state of the blockchain and it takes some time for their transaction to be included, they do not step into the next epoch. As long as k is chosen wisely, transactions will be processed before the account changes state.

Rolling over on a smart contract. Unfortunately, rolling over is not as straightforward as it may appear because a smart contract does not do anything unless a transaction is sent to it. One cannot expect every user to send a rollover message for every epoch. Moreover, there is no way they will be able to get such a message in at the right time.

A first idea is to 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.

We rollover an account in an epoch when the first message from *this* account is received; so, one message rolls over only one account. To achieve this, 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.

Replay protection. The mechanism we have described so far *almost* works for transferring confidentially, except for the problem of replay attacks. 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 due to two reasons: (1) Zether accounts have their own public keys; they are not associated with Ethereum addresses. (2) Zether transactions contain *non-interactive* ZK-proofs. A malicious actor can steal these proofs and put them inside 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. Later in this section, we will discuss another important feature of Zether: locking accounts to other smart contracts. Even if the request to lock a Zether account shows knowledge of the secret key, and the account is unlocked later, an adversarial entity could lock the account again by just replaying the request.

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.

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.

3.2 Transferring Anonymously

Our next goal is to enable users of Zether to transfer ZTH anonymously. This creates a number of new and interesting issues, which we elaborate below.

More complex ZK-proof. An anonymous transfer transaction enables a user Alice with public key y_A interested in sending b ZTH to another user Bob with public key y_B to hide both herself and Bob among a larger group of users $\mathbb{U}$ with public keys $\{y_1, \ldots, y_n\}$, along with hiding b. ($\mathbb{U}$ of course contains Alice and Bob.) As a result, Alice provides a more complex ZK-proof π. Specifically, Alice generates n ciphertexts $C_1, \ldots, C_n$ under keys $y_1, \ldots, y_n$ respectively. π shows that all the ciphertexts are well-formed; only two of the underlying amounts are non-zero and their values only differ in sign; and, the remaining balance of the account with positive amount is non-negative.

   We use several optimizations to reduce the size of π and verification cost. The prover sends n ciphertexts, and all of them expect two encrypt 0. We leverage this in our proof construction such that the proof statement only contains two range proofs, as sub-statements. Concretely we use one-out-of-many proofs [41]. 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 $-b$ under y_A and y_B respectively along with a secondary encryption of Alice's balance b^*. Alice then simply shows the relationship between b and $-b$ and that b and $b^* - b$ are non-negative using a range proof.

Replay and double-spend protection. There is a subtle issue with using the same replay protection mechanism we proposed earlier for confidential transfers. An anonymous transaction published by Alice involves multiple accounts only one of which Alice may own. To preserve anonymity, we cannot treat Alice's account in any special way. Indeed, 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. Unfortunately, this immediately leads to a serious problem. Other account holders involved in Alice's transaction may have generated a transaction with the previous value of nonce. 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 transfer 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 transfer 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.

3.3 Interoperability with Smart Contracts

A main design goal for Zether is to be interoperable with other arbitrary smart contracts. This presents a challenge as it is impossible to know in which way smart contracts will want to use Zether. Further, the other smart contracts may contain bugs or even be maliciously designed. Unlike a cleartext currency like ETH, a smart contract cannot simply issue a ZTH transfer as doing so requires creating ZK-proofs. Smart contracts, however, have only public states and thus are unable to store secrets necessary to create ZK-proofs.

Our solution is to *lock* an account to a smart contract, say SC. In effect, this transfers the ownership of the account to SC. Any transfer from the account will need to go through SC. The transfer still needs to contain a valid ZK-proof, and as such will need to be created by the user. SC finally can *unlock* an account to return control to the original user. In Section 11, we demonstrate the versatility of this design on multiple applications. Though the locking mechanism is simple, we need to consider some subtleties.

Anonymous transfers. If some accounts involved in an anonymous transaction are locked to a smart contract then, naturally, all of the locked accounts must be locked to the same contract. Furthermore, the transaction is processed only if it comes from that contract.

Pending locks. Suppose Alice publishes a transaction in a certain epoch to lock her account to a smart contract. Should locking come into effect immediately? Actually, it should *not* because 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. (Recall that every method on ZSC first checks if there is a need for roll over.)

Replay attacks. 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.

3.4 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 [16] with Σ protocols to enable efficient proofs on algebraically-encoded values. Concretely, we can efficiently prove that a set of ElGamal encrypted values are in some range. Further, we can combine one-out-of-many proofs also known as ring signatures with range proofs to allow anonymous transfers. Σ-Bullets inherit 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.

4 Preliminaries and Definitions

4.1 Notation

We use λ to denote the security parameter. Let 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 $\{a, a+1, \ldots, b-1, b\}$.

We use $x \xleftarrow{\$} S$ to denote 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.

4.2 Tools

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 $r \xleftarrow{\$} \mathbb{Z}_p^\star$. 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

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

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

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 [2].

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.

We represent a signature scheme with algorithms $(\mathsf{Setup}_{\mathsf{sig}}, \mathsf{Sign}, \mathsf{Verify}_{\mathsf{sig}})$, where $\mathsf{Setup}_{\mathsf{sig}}$ outputs some public parameters, Sign generates a signature on an input message, and $\mathsf{Verify}_{\mathsf{sig}}$ 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 [10] for a thorough treatment of the properties.

5 Payment Mechanism

5.1 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.[2]

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 the applications section 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 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 2 and Figure 4, where we describe the details of our Zether instantiation, we use a programming oriented, Solidity inspired, syntax as defined in Table 1.)

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 $\mathsf{tx}_{\mathsf{fund}} = (\mathsf{pk}, \ldots)$.

3. $\mathsf{CreateTransferTx}(\mathsf{sk}_{\mathsf{from}}, \mathsf{pk}_{\mathsf{to}}, \mathsf{AnonSet}, \mathsf{amt}, \mathsf{st}_h) \to \mathsf{tx}_{\mathsf{trans}}$. $\mathsf{CreateTransferTx}$ is used to transfer money from one account to another amongst a set of accounts. It takes a secret

---

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

key sk_{from}, a destination public key pk_{to}, a set of public keys AnonSet such that both $pkOf(sk_{from})$ and pk_{to} belong to it, an amount amt, and the state of the smart contract st_h at a certain block height h as inputs. It outputs $tx_{trans} = (AnonSet, \ldots)$.

4. $CreateBurnTx(sk, st_h) \to tx_{burn}$. 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 $tx_{burn} = (pkOf(sk), amt, \ldots)$.

5. $CreateLockTx(sk, addr, st_h) \to tx_{lock}$. 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 $tx_{lock} = (pkOf(sk), addr, \ldots)$.

6. $CreateUnlockTx(pk) \to tx_{unlock}$. CreateUnlockTx is used to unlock an account. It takes a public key pk as input. It outputs $tx_{unlock} = (pk, \ldots)$.

7. $ReadBalance(sk, st_h) \to b$. 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.

Smart contract. The smart contract has five functions Fund, Transfer, Burn, Lock and Unlock. They take $tx_{fund}, tx_{trans}, tx_{burn}, tx_{lock}$ and tx_{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 $E \geq 1$. The epoch number of a block at height h is defined to be $\lfloor h/E \rfloor$. Thus, for example, the blocks at heights $0, 1, \ldots, E - 1$ are in the first epoch, the ones at heights $E, E + 1, \ldots, 2E - 1$ are in the second epoch, and so on.

Honest behavior. Three of the (transaction-generating) user algorithms described above, transfer, burn and lock, take a state of the blockchain as input. An honest user is supposed to run these algorithms only at the beginning of an epoch, using the state of the epoch that just concluded. Indeed, the user may not be in complete sync with the blockchain and a transaction could suffer some delay, but a good choice of epoch length would take care of these problems. Also, no more than one transfer or burn transaction (not one of each) should be generated per account in any epoch. (This restriction goes away if we only care about confidentiality, see Section 7.) For a transfer transaction, if user puts a locked account in its anonymity set, then all accounts must be locked to the same address.

5.2 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. At the beginning of epoch e_1, she publishes a transfer transaction to send Y ZTH to someone else. There could be other users in the network who include

Alice's account in their anonymity set. Their transactions could be processed before Alice's transaction and so on. In e_1, Alice also receives Z ZTH from others. Now, in any epoch after e_1, if she publishes a burn transaction at the start of the epoch with amount $X - Y + Z$, then she should 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. For e.g., for a transfer transaction, it is checked that if some accounts in the anonymity set are locked, then they are all locked to the same Ethereum address. 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*. See Appendix B for a formal description of the correctness requirement.

5.3 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. For example, no adversary should be able to distinguish between transfer transactions that transfer *any* amount of money between *any* two public keys in an anonymity set which consists of honest parties only, as long as the sender has enough money. The formulation of overdraft-safety and privacy is inspired by Zerocash [3].

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. For e.g., if the anonymity sets associated with two create transfer transaction instructions are different, then Adv can trivially guess b.

See Appendix B for a formal description of the security requirements.

6 Zether Payment Mechanism

We now present a secure payment mechanism called Zether. It 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 C 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.

6.1 Setup

The setup algorithm calls Setup_π and $\mathsf{Setup}_{\mathsf{sig}}$ 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 10), we use Bulletproofs [16] and Schnorr signatures [74], both of which have a trustless setup. Zether significantly differs from Zerocash [3] in this respect because Zerocash has a trusted setup and its security is broken if the setup is subverted.

Zether's setup specifies an epoch length E and a maximum amount value MAX. Honest users are supposed to make only one transfer or burn transaction per epoch. If these transactions are delayed by more than E, then they may become invalid. Thus E must be carefully chosen (see Section 3.1 for a discussion).

Setup algorithm is formally described in Figure 1. Apart from setting up proof system and signature scheme, it initializes account tables acc and pending transfers table pTransfers (recall that incoming transfers are put into a pending state first), a last roll over epoch table 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 pending lock table pLock to temporarily hold lock requests, a nonce set $\mathcal{U}$ to prevent replay attacks, a variable b_{total} that tracks the total amount of ZTH held by the contract, a global last update variable t to record the last epoch in which *any* account was updated, and, lastly, an epoch base g_{epoch}.

6.2 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

INPUT: Security parameter λ (in unary)
1. $(p, g, \mathbb{G}) \leftarrow \mathsf{GroupGen}(1^\lambda)$
2. $\mathrm{pp}_\pi \leftarrow \mathsf{Setup}_\pi(1^\lambda)$
3. $\mathrm{pp}_{\mathsf{sig}} \leftarrow \mathsf{Setup}_{\mathsf{sig}}(1^\lambda)$
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 pending lock table, $\mathsf{pLock} : \mathbb{G} \to \{0,1\}^*$,
   - an empty nonce set, $\mathcal{U} = \emptyset$,
   - total balance $b_{\mathsf{total}} \in \mathbb{Z}$ to 0,
   - global last update in epochs $t \in \mathbb{Z}$ to 0,
   - first epoch base $g_{\mathsf{epoch}} \in \mathbb{G}$ to $\mathcal{H}(\texttt{Zether}\|0)$, where $\mathcal{H}$ is a hash function.
5. Deploy smart contract ZSC (Figure 2) with parameters $(p, g, \mathbb{G}), \mathrm{pp}_\pi, \mathrm{pp}_{\mathsf{sig}}, \mathsf{acc}, \mathsf{pTransfers},$
   $\mathsf{lastRollOver}, \mathsf{lock}, \mathsf{pLock}, \mathcal{U}, b_{\mathsf{total}}, t, g_{\mathsf{epoch}}, \mathsf{MAX}, \mathsf{E}.$

Figure 1: Zether set-up.

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:

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

The statement shows that the user knows an sk such that y is indeed the public key corresponding to sk, (C_L, C_R) is a valid encryption of b under y, and u is the correct nonce w.r.t. sk for the current epoch. A simple Σ protocol can be used to prove the statement.

Confidential Transaction. Let us now consider a transfer transaction. For intuition, we first describe the language of valid transactions in the purely confidential variant of Zether which does not provide any additional anonymity. This variant is described in detail later (Section 7) and is simpler. 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 in the confidential version of Zether:

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

Kurosawa [48] 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 $\overline{y}$.

Anonymous Transaction. Anonymous transfers not only hide the transfer amount but also the sender and receiver in a transfer. Like above, a user wants to transfer an amount $b^\star$ from a public key y to $\overline{y}$, but now it would like to hide the two keys among a larger set of public keys, say $\mathbf{y} = \{y_1, \ldots, y_n\}$. Let $(C_{L,i}, C_{R,i})$ be the encryption of the balance associated with key y_i, for $i \in [n]$. The user now needs to create n ciphertexts $(C_1, D_1), \ldots, (C_n, D_n)$ and prove that:

- they are well-formed;
- one of them (say jth) encrypts $b^\star$, one of them (say ℓth) encrypts $-b^\star$, rest encrypt 0;
- $b^\star$ is positive;
- remaining balance in y_j (say b') is also positive; and,
- the correct nonce is declared.

We can again reuse randomness and set $D_1 = \cdots = D_n = D$.

To handle such a complex statement efficiently without revealing j, ℓ, $b^\star$ and b', we introduce new *binary* variables $s_1, \ldots, s_n$ and $t_1, \ldots, t_n$. Value 1 for an s_i denotes that money is being transferred *from* y_i and value 1 for a t_j denotes that money is being transferred *to* y_j. The user would keep these variables secret and prove various statements using them. Only one of $s_1, \ldots, s_n$ and one of $t_1, \ldots, t_n$ should be 1. This could be shown by proving that each of these variables is either 0 or 1, $\sum_i s_i = 1$ and $\sum_i t_i = 1$.

In addition, the user proves:

$$\prod_{i=1}^{n} C_i^{s_i} = g^{b^\star} \prod_{i=1}^{n} y_i^{r \cdot s_i} \tag{3}$$

$$\prod_{i=1}^{n} C_i^{s_i + t_i} = \prod_{i=1}^{n} y_i^{r \cdot (s_i + t_i)} \tag{4}$$

$$C_i^{(1-s_i) \cdot (1-t_i)} = y_i^{(1-s_i) \cdot (1-t_i) \cdot r} \text{ for } i \in [n] \tag{5}$$

$$\prod_{i=1}^{n} \left(\frac{C_{L,i}}{C_i} \right)^{s_i} = g^{b'} \left(\frac{\prod_{i=1}^{n} C_{R,i}^{s_i}}{D} \right)^{\mathsf{sk}} \tag{6}$$

$$g_{\mathsf{epoch}}^{\mathsf{sk}} = u \tag{7}$$

Given that exactly one of $s_1, \ldots, s_n$ is 1 and rest are 0, Eq (3) shows that the ciphertext for non-zero s_i is a valid encryption of $b^\star$. When Eq (3) is subtracted from Eq (4), we get $\prod_{i=1}^{n} C_i^{t_i} = g^{-b^\star} \prod_{i=1}^{n} y_i^{r \cdot t_i}$, which shows that the ciphertext for non-zero t_i is a valid encryption of $-b^\star$. Thus, Eq (3) and (4) together show that the ciphertexts that encode non-zero quantities are valid.

These statements can be proved more efficiently using techniques from one-out-of-many proofs [41] and in particular the extension to ElGamal encryptions [13]. These proofs are Σ protocols that can be used to show that a decryption of one out of n ciphertexts has certain properties, e.g. they are 0. The proof size is only logarithmic in n. This is achieved by writing i in it's binary representations and constructing n products such that only the ith is 1 and all other are 0.

In Eq (5), the expression $(1 - s_i)(1 - t_i)$ is non-zero only when both s_i and t_i are zero. For such i, C_i should be an encryption of 0, which is what the equation shows. (When s_i or t_i is non-zero, both sides are of the equation are just 0.) Eq (6) shows that b' balance is left in the account for which s_i is 1. Lastly, Eq (7) shows that u is the correct nonce for the current epoch.

Further, the user also needs to show that $g^{\mathsf{sk}} = \Pi y_i^{s_i}$, $b^\star \in [0, \mathsf{MAX}]$ and $b' \in [0, \mathsf{MAX}]$, where the first equation ties the secret key to the spending public key (without revealing the latter), and the rest two show that the amount being transferred and the remaining amount are in the right range. Summing it up, the user proves the following statement:

$\mathsf{st}_{\mathsf{AnonTransfer}}$:
$$\Big\{ ((y_i, C_{L,i}, C_{R,i}, C_i)_{i=1}^n, D, u, g, g_{\mathsf{epoch}}; \mathsf{sk}, b^\star, b', r, (s_i, t_i)_{i=1}^n) :$$
$$\prod_{i=1}^n C_i^{s_i} = g^{b^\star} \prod_{i=1}^n y_i^{r \cdot s_i} \wedge \prod_{i=1}^n C_i^{s_i + t_i} = \prod_{i=1}^n y_i^{r \cdot (s_i + t_i)} \wedge$$
$$D = g^r \wedge (C^{(1-s_i) \cdot (1-t_i)} = y_i^{(1-s_i) \cdot (1-t_i) \cdot r})_{i=1}^n \wedge$$
$$\prod_{i=1}^n \left(\frac{C_{L,i}}{C_i} \right)^{s_i} = g^{b'} \left(\frac{\prod_{i=1}^n C_{R,i}^{s_i}}{D} \right)^{\mathsf{sk}} \wedge g^{\mathsf{sk}} = \prod_{i=1}^n y_i^{s_i} \wedge$$
$$g_{\mathsf{epoch}}^{\mathsf{sk}} = u \wedge (s_i \in \{0,1\} \wedge t_i \in \{0,1\})_{i=1}^n \wedge$$
$$\sum_{i=1}^n s_i = 1 \wedge \sum_{i=1}^n t_i = 1 \wedge b^\star \in [0, \mathsf{MAX}] \wedge b' \in [0, \mathsf{MAX}] \Big\} \quad (8)$$

The statement is complex, but in fact highly structured. The resulting proof size can be logarithmic in the range and in the anonymity set. This is done by combining one-out-of-many proofs for showing that two encryptions are non-zero with Bulletproofs for range proofs and for showing that all other encryptions encrypt 0. The proofs can be efficiently combined using our extension of Bulletproofs called Σ-Bullets (Section 9). Even with this efficient proof the transaction size will still be linear in the anonymity set. This is because the sender needs to send an encryption for each account in the anonymity set. This seems inherent as every account needs to be treated equally in an anonymous transfer. This limitation is not present in UTXO-based currencies where constant sized fully anonymous payment systems have been proposed and implemented [3, 56]. Theoretically, the requirement could be removed using fully homomorphic encryption but it remains an open problem whether a more practical solution is available.

6.3 Zether Contract

The Zether contract (ZSC) is defined in Figure 2. 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; see Table 1 for the syntax we use and what it means. 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

block.number	current block number
msg.sender	sender of the message (current call)
msg.value	number of Wei sent with the message
address.transfer	send given amount of Wei to address
require	check conditions; current call reverts if condition fails

Table 1: Solidity syntax

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 the last roll over epoch. It also rolls over any changes to the locking address of y. Some global changes also need to be made in an epoch. For this, RollOver checks if the last global update t was in a older epoch. If so, then it sets g_{epoch} for the current epoch, empties the nonce set $\mathcal{U}$ and resets t.

Locking. We next discuss the methods related to locking. Recall that (un)locking is not supposed to take effect immediately, i.e. in the same epoch as the transaction is published; it must be deferred to the next epoch (see Section 3.3). 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 epoch base g_{epoch}. (This ensures that the account holder indeed wants to lock the account to addr at this time. Note that even if the signature is replayed, it would not affect anything.) It then sets addr to be a pending lock on y. Unlock method also calls CheckLock first, then sets the pending lock to be $\bot$. In a later epoch, when RollOver is called, it will set the lock to be the value in pending lock.

The internal method CheckLock takes as input a public key y and an Ethereum address addr, and returns 1 if it is appropriate for addr to operate on y. This is possible in two situations: a) if lock is $\bot$ meaning that any address could operate on y or b) if lock is equal to addr.

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 given proof π_{burn} (see $\mathsf{st}_{\mathsf{burn}}$ in (1)) and adds the nonce u to the set $\mathcal{U}$. The proof cannot be replayed in the same epoch because Burn also checks that $u \notin \mathcal{U}$. It also cannot be replayed in any other epoch because the epoch base would be different. Note that a burn operation does not close an account.

Transfer. Transfer transfers some ZTH from one of the accounts in the set $\mathbf{y}$ to another account. Since the transfer is anonymous, all accounts are treated in the same way; some (encrypted) balance is added to each one of them. The proof π_{transfer} makes sure that these balances satisfy the right properties (see $\mathsf{st}_{\mathsf{AnonTransfer}}$ in (8)).

6.4 User Algorithms

User algorithms specify how users can interact with ZSC. They are described in Figure 3. The algorithms to create transfer and burn transactions first do a roll over of the input public key(s). This ensures that any pending transfers are rolled over and any pending lock request takes effect. In CreateTransferTx, the commitments C_i and the variables s_i, t_i for the sender and receiver (indices i_{from} and i_{to}) are set to correspond to non-zero values. 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.

7 Basic Zether

Zether provides both anonymity and confidentiality. As discussed in the overview (Section 3), anonymity impacts both efficiency and usability: ZK-proofs are more complex and only one transfer/burn transaction can be made from an account in an epoch. Therefore, we also provide a description of a non-anonymous Zether which provides just confidentiality but is more efficient. We refer to this version of Zether as basic Zether.

In basic Zether, replay protection is achieved by associating a counter with every account. With every transaction, users need to provide a signature on the transaction data and the counter, binding the two together. (Ethereum takes the same approach.) Note that there would not be any restriction now on the number of transfer or burn transactions.

Global set-up. There are three main differences in set-up. First, there is no need to initialize a nonce-set, an epoch base or a global last update. Second, a new counter table $\mathsf{ctr} : \mathbb{G} \to \mathbb{Z}$ is initialized to empty. And, third, there is only a single lock table lock instead of two before.

Smart contract. We go through the contract methods in basic Zether one by one, highlighting the differences from Zether. There is only one change in the method Fund: when a new record is created for y, the corresponding counter is set to 0. In methods Burn and Transfer, any logic related to nonces goes away. Instead, they take a signature as input and verify that it properly signs all the inputs and the counter. A simpler statement for transfer, $\mathsf{st}_{\mathsf{ConfTransfer}}$ (see Eq (2)), needs to be proved now. The statement for burn is also slightly simplified (but at the cost of an additional signature). $\mathsf{st}'_{\mathsf{burn}}$ is the same as $\mathsf{st}_{\mathsf{burn}}$ except that the check $u = g_{\mathsf{epoch}}^{\mathsf{sk}}$ is removed.

In method Lock, the signature now needs to be on the counter instead of the epoch base. Locking and unlocking can go into effect immediately. RollOver is slightly simplified now because no global changes need to be made (the second if condition goes away) and no logic related to locking needs to be handled. See Figure 4 for a formal description of the smart contract methods.

<u>Fund</u>
- INPUTS: public key y
1. RollOver(y)
2. Let $b = \mathsf{msg.value}$
3. require:
   - $b + b_{\mathsf{total}} \leq \mathsf{MAX}$
   - CheckLock($y, \mathsf{msg.sender}$) = 1
4. If acc[y] = $\perp$:
   - Let $H = \mathsf{block.number}$, $e = \lfloor H/\mathsf{E} \rfloor$
   - Set acc[y] = $(1,1)$
   - Set pTransfers[y] = $(g^b, 1)$
   - Set lock[y] = pLock[y] = $\perp$
   - Set lastRollOver[y] = e

   Else:
   - Set pTransfers[y] = pTransfers[y] $\circ (g^b, 1)$
5. Set $b_{\mathsf{total}} = b_{\mathsf{total}} + b$

<u>Transfer</u>
- INPUTS:
   - set of public keys $\mathbf{y} = \{y_1, \ldots, y_n\}$
   - ciphertexts $(C_1, D), (C_2, D), \ldots, (C_n, D)$
   - nonce u
   - proof π_{transfer}
1. For each $i \in [1, n]$:
   a. RollOver(y_i)
   b. Let $(C_{L,i}, C_{R,i}) = $ acc[y_i]
2. Let $\mathbf{C} = \{C_1, \ldots, C_n\}$
3. require:
   - $u \notin \mathcal{U}$
   - CheckLock($y_i, \mathsf{msg.sender}$) = 1 for $i \in [1, n]$
   - $\mathsf{Verify}_{\mathsf{nizk}}(\mathsf{st}_{\mathsf{AnonTransfer}}[\mathbf{y}, \mathbf{C}, D,$
     $(C_{L,i}, C_{R,i})_{i=1}^n, u, g, g_{\mathsf{epoch}}], \pi_{\mathsf{transfer}}) = 1$
4. Set $\mathcal{U} = \mathcal{U} \cup \{u\}$
5. For each $i \in [1, n]$, set
   pTransfers[y_i] = pTransfers[y_i] $\circ (C_i, D)$

<u>Lock</u>
- INPUTS:
   - public key y
   - Ethereum address addr
   - signature σ_{lock}
1. RollOver(y)
2. require:
   - CheckLock($y, \mathsf{msg.sender}$) = 1
   - $\mathsf{Verify}_{\mathsf{sig}}(y, (\mathsf{addr}, g_{\mathsf{epoch}}), \sigma_{\mathsf{lock}}) = 1$
3. Set pLock[y] = addr

<u>Burn</u>
- INPUTS:
   - public key y
   - balance b
   - nonce u
   - proof π_{burn}
1. RollOver(y)
2. Let $(C_L, C_R) = $ acc[y]
3. require:
   - $u \notin \mathcal{U}$
   - CheckLock($y, \mathsf{msg.sender}$) = 1
   - $\mathsf{Verify}_{\mathsf{nizk}}(\mathsf{st}_{\mathsf{burn}}[y, C_L, C_R, u, b, g, g_{\mathsf{epoch}}],$
     $\pi_{\mathsf{burn}}) = 1$
4. Set $\mathcal{U} = \mathcal{U} \cup \{u\}$
5. Set acc[y] = acc[y] $\circ (C_L^{-1}, C_R^{-1})$
6. Set $b_{\mathsf{total}} = b_{\mathsf{total}} - b$
7. Do $\mathsf{msg.sender.transfer}(b)$

<u>Unlock</u>
- INPUTS: public key y
1. RollOver(y)
2. require:
   - CheckLock($y, \mathsf{msg.sender}$) = 1
3. Set pLock[y] = $\perp$

Internal Helper Methods

<u>RollOver</u>
- INPUTS: public key y
1. Let $H = \mathsf{block.number}$, $e = \lfloor H/\mathsf{E} \rfloor$
2. If lastRollOver[y] < e:
   - Set acc[y] = acc[y] $\circ$ pTransfers[y]
   - Set pTransfers[y] = $(1, 1)$
   - Set lock[y] = pLock[y]
   - Set lastRollOver[y] = e
3. If $t < e$:
   - Set $g_{\mathsf{epoch}} = \mathcal{H}(\mathtt{Zether}\|e)$
   - Set $\mathcal{U} = \emptyset$
   - Set $t = e$

<u>CheckLock</u>
- INPUTS:
   - public key y
   - Ethereum address addr
- OUTPUT: 1 if account y can be operated by addr; 0 otherwise
1. If lock[y] = $\perp$ or lock[y] = addr:
   - Output 1

   Else:
   - Output 0

Figure 2: The Zether smart contract ZSC.

CreateAddress
- INPUTS: 1^λ
- OUTPUT: $x \in \mathbb{Z}_p, y \in \mathbb{G}$
1. $x \xleftarrow{\$} \mathbb{Z}_p$
2. $y = g^x$

CreateTransferTx
- INPUTS:
  - shuffled anonymity set $\mathbf{y} = \{y_1, \ldots, y_n\}$ which includes sender and receiver
  - sender index i_{from}
  - receiver index i_{to}
  - sender private key x
  - sender balance b_{from}
  - transfer amount $b^\star$
  - state st_h of ZSC
- OUTPUT: $\mathsf{tx}_{\mathsf{trans}} = (\mathbf{y}, \mathbf{C}, D, u, \pi_{\mathsf{transfer}})$
1. Roll over all y_i in st_h
2. Let $(C_{L,i}, C_{R,i}) = \mathsf{acc}[y_i]$ for $i \in [n]$
3. $r \xleftarrow{\$} \mathbb{Z}_p$
4. Set $C_{i_{\mathsf{from}}} = g^{b^\star} y_{i_{\mathsf{from}}}^r$
5. Set $C_{i_{\mathsf{to}}} = g^{b^\star} y_{i_{\mathsf{to}}}^r$
6. Set $C_i = y_i^r$ for $i \in [n] \setminus \{i_{\mathsf{from}}, i_{\mathsf{to}}\}$
7. Set $\mathbf{C} = \{C_1, \ldots, C_n\}$ and $D = g^r$
8. Set $s_{i_{\mathsf{from}}} = 1$ and $s_i = 0$ for $i \in [n] \setminus \{i_{\mathsf{from}}\}$
9. Set $t_{i_{\mathsf{to}}} = 1$ and $t_i = 0$ for $i \in [n] \setminus \{i_{\mathsf{to}}\}$
10. Set $u = g_{\mathsf{epoch}}^x$
11. Set $w = (x, b^\star, b_{\mathsf{from}} - b^\star, r, (s_i, t_i)_{i=1}^n)$
12. $\pi_{\mathsf{transfer}} =$
    $\mathsf{Prove}(\mathsf{st}_{\mathsf{AnonTransfer}}[\mathbf{y}, \mathbf{C}, D, (C_{L,i}, C_{R,i})_{i=1}^n, u, g, g_{\mathsf{epoch}}], w)$

CreateBurnTx
- INPUTS:
  - private key x
  - state st_h of ZSC
- OUTPUT: $\mathsf{tx}_{\mathsf{burn}} = (y, b, u, \pi_{\mathsf{burn}})$
1. Let $b = \mathsf{ReadBalance}(x, \mathsf{st}_h)$
2. $u = g_{\mathsf{epoch}}^x$
3. $w = (x)$
4. Set $y = g^x$
5. Let $(C_L, C_R) = \mathsf{acc}[y]$
6. $\pi_{\mathsf{burn}} =$
   $\mathsf{Prove}(\mathsf{st}_{\mathsf{burn}}[y, C_L, C_R, u, b, g, g_{\mathsf{epoch}}], w)$

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 $\sigma_{\mathsf{lock}} = \mathsf{Sign}(x, (\mathsf{addr}, g_{\mathsf{epoch}}))$
2. Set $y = g^x$

ReadBalance
- INPUTS:
  - private key x
  - 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. $C_L/C_R^x = g^b$

Figure 3: User algorithms. CreateFundTx and CreateUnlockTx algorithms are quite straightforward, so they are not specified here.

User algorithms. CreateAddress, CreateFundTx, CreateUnlockTx and ReadBalance do not change. CreateBurnTx and CreateTransferTx produce proofs for different statements now and sign the output with the counter. CreateLockTx signs the counter instead of the epoch base. See Figure 5 for a formal description of the algorithms.

8 Discussion and Extensions

We discuss here certain important things related to the use of Zether.

Paying gas in ZTH through economic abstraction. One of the main limitations of the anonymity of Zether comes from the fact that every transaction needs to pay fees in ETH pegged gas. This is not only cumbersome as a ZTH holder needs to maintain both a Zether and an Ethereum account, it can also have consequences for the anonymity of a transaction.

Fund
- INPUTS: public key y
1. RollOver(y)
2. Let $b = $ msg.value
3. require:
    - $b + b_{\text{total}} \leq$ MAX
    - CheckLock(y, msg.sender) $= 1$
4. If acc$[y] = \perp$:
    - Let $H = $ block.number, $e = \lfloor H/\text{E} \rfloor$
    - Set acc$[y] = (1, 1)$
    - Set pTransfers$[y] = (g^b, 1)$
    - Set lock$[y] = \perp$
    - Set lastRollOver$[y] = e$
    - Set ctr$[y] = 0$
    Else:
    - Set pTransfers$[y] = $ pTransfers$[y] \circ (g^b, 1)$
5. Set $b_{\text{total}} = b_{\text{total}} + b$

Transfer
- INPUTS:
    - sender address y
    - recipient address $\overline{y}$
    - ciphertexts (C, D), $(\overline{C}, D)$
    - proof π_{transfer}
    - signature σ_{transfer}
1. RollOver(y)
2. Let $(C_L, C_R) = $ acc$[y]$
3. require:
    - CheckLock(y, msg.sender) $= 1$
    - Verify$_{\text{nizk}}$(st$_{\text{ConfTransfer}}[y, \overline{y}, C_L, C_R, C, \overline{C}, D]$, $\pi_{\text{transfer}}) = 1$
    - Verify$_{\text{sig}}(y, (\overline{y}, C, \overline{C}, D, \pi_{\text{transfer}}, \text{ctr}[y])$, $\sigma_{\text{transfer}}) = 1$
4. Set acc$[y] = $ acc$[y] \circ (C^{-1}, D^{-1})$
5. Set pTransfers$[\overline{y}] = $ pTransfers$[\overline{y}] \circ (\overline{C}, D)$
6. Set ctr$[y] = $ ctr$[y] + 1$

Lock
- INPUTS:
    - public key y
    - Ethereum address addr
    - signature σ_{lock}
1. RollOver(y)
2. require:
    - CheckLock(y, msg.sender) $= 1$
    - Verify$_{\text{sig}}(y, (\text{addr}, \text{ctr}[y]), \sigma_{\text{lock}}) = 1$
3. Set lock$[y] = $ addr
4. Set ctr$[y] = $ ctr$[y] + 1$

Burn
- INPUTS:
    - public key y
    - balance b
    - proof π_{burn}
    - signature σ_{burn}
1. RollOver(y)
2. Let $(C_L, C_R) = $ acc$[y]$
3. require:
    - CheckLock(y, msg.sender) $= 1$
    - Verify$_{\text{nizk}}$(st$'_{\text{burn}}[y, C_L, C_R, b, g], \pi_{\text{burn}}) = 1$
    - Verify$_{\text{sig}}(y, (b, \pi_{\text{burn}}, \text{ctr}[y]), \sigma_{\text{burn}}) = 1$
4. Set acc$[y] = $ acc$[y] \circ (C_L^{-1}, C_R^{-1})$
5. Set ctr$[y] = $ ctr$[y] + 1$
6. Set $b_{\text{total}} = b_{\text{total}} - b$
7. Do msg.sender.transfer(b)

Unlock
- INPUTS: public key y
1. RollOver(y)
2. require:
    - CheckLock(y, msg.sender) $= 1$
3. Set lock$[y] = \perp$

Internal Helper Methods

RollOver
- INPUTS: public key y
1. Let $H = $ block.number, $e = \lfloor H/\text{E} \rfloor$
2. If lastRollOver$[y] < e$:
    - Set acc$[y] = $ acc$[y] \circ $ pTransfers$[y]$
    - Set pTransfers$[y] = (1, 1)$
    - Set lastRollOver$[y] = e$

Figure 4: Smart contract functions in basic Zether. CheckLock is same as in Figure 2 so not defined here.

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 in st_h
2. Let $(C_L, C_R) = \mathsf{acc}[y]$
3. $r \xleftarrow{\$} \mathbb{Z}_p$
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 $y = g^x$

Figure 5: User algorithms in basic Zether. CreateAddress, CreateFundTx, CreateUnlockTx and ReadBalance are same as in Figure 3.

[3] Every transaction is initiated by an EOA (see Section 2) and that EOA pays for the gas. Unfortunately, no matter what privacy guarantees Zether has, transactions that are issued by the same EOA are inevitably linked. Even if a user controls multiple EOAs she has to make sure that they remain unlinkable in order to send anonymous transactions on Zether. (The confidentiality of a transaction is not affected by this.)

Ideally, miners could receive transaction fees in ZTH instead of ETH which are by design exchangeable and have as such identical value. In order to do this, a transaction will be sent with a 0 gas price. This means that a newly created unlinked EOA can issue the transaction. Every transaction will specify the fee amount publicly. In the ZK-proof the sender now proves that his new balance is his old balance minus the transacted amount minus the fees. The fees is then sent to a fee pool within the Zether contract that can be spent from without further permission or secrets. A miner can simply insert a raking transaction at the end of the block that pays all of the fees to his account. The fee pool could also be used to fund contract-wide transactions such as RollOver.

The idea of paying fees in a non-native currency is called economic abstraction and has been discussed intensively [18, 72]. The concept is particularly interesting with respect to Zether as it is would make Zether a) more usable and b) more private. The major obstacle to this approach is that miners would need to mine these special 0 gas price transactions and properly rake the fee pool. A similar approach that circumvents the miner adoption problem

---

[3]Anonymity is a hard to achieve goal in general. There are attacks on Monero [47, 58] and Zcash [44, 67]. Even if an anonymous network like Tor [81] is used, cryptocurrencies can be attacked [6].

is to have special delegator nodes that issue the transactions to the network. Users would send their transactions to delegators who will forward them to the miners and will pay the Ethereum gas fees. These delegator nodes could be rewarded in Zether by adding their Zether address to the transaction. The fee amount would not go to the fee pool but to that address.

A general problem for any anonymous blockchain transaction is that transactions need to be relayed to either the miners or the delegators without revealing the original sender's identity. This problem can be alleviated through anonymous communication networks like Tor [81].

Zether as a native token. Zether can be viewed as a mechanism to do private transactions in an account-based currency. If a currency like Ethereum [17], Ripple [70], EOS [33] or Stellar [53] or even a private blockchain like Sequence [76] wanted to introduce private transactions then they can use a Zether style system. While we have presented Zether as a smart contract token that can be run on a smart contract platform, it is also possible to envision Zether becoming the native token of such a platform. Using Zether as a native token has several advantages. As discussed in the previous paragraph, it is important for Zether's anonymity that miners receive fees in ZTH. This is automatically ensured if ZTH is the native token in which fees are denominated. An additional benefit of Zether as a native token is that denial-of-service protection becomes simpler. Miners can process transactions in an arbitrary order. For example the proofs can be verified *before* the account balances are transferred. We can potentially remove the requirement of having separate stable and temporary accounts as the ordered processing allows a user to spend ZTH even if they receive funds in the same block. Depending on the block-time, it may still be useful to maintain a temporary and a stable account to allow slightly outdated transactions to be processed correctly.

In Section 10.3, we discuss multiple optimizations that would benefit Zether. If Zether were a native token then these optimizations can be used to make transactions more efficient and scalable. For example, miners can run optimized proof validation software using efficient elliptic curves instead of using Ethereum's general purpose state machine. Further, Bulletproofs used in our instantiation of Zether can be verified more efficiently when processed in batches [16]. This is beneficial as a fully verifying node can now more efficiently process a proposed block. This property is important in order to protect against the miners' dilemma [37], where miners are disincentivized to verify blocks if verification becomes too expensive.

Multiple recipients. Zether transfer transactions have one recipient only, but quite often Alice may want to send ZTH to several other users at the same time. While Alice can do a separate transfer for every recipient, we can define a new transfer method that accomplishes the same task much more efficiently. Let us consider the simpler case of confidential transactions and recall the statement $\mathsf{st}_{\mathsf{ConfTransfer}}$ (2) from Section 6.2. Suppose Alice has a balance b and she wants to transfer $b_1^\star, \ldots, b_n^\star$ to n users.

We know that range proofs are the most expensive components of ZK-proofs in Zether both in terms of communication and computation. If separate transactions are generated, then Alice needs to show a total of $2n$ values to be positive: $b_1^\star$, $b - b_1^\star$, $b_2^\star$, $(b - b_1^\star) - b_2^\star$, ..., $b_n^\star$, $(b - b_1^\star - \ldots - b_{n-1}^\star) - b_n^\star$. Instead, we could generate a single transaction that consists of an encryption of $\hat{b} = b_1^\star + \ldots + b_n^\star$ under Alice's public key, encryptions of $b_i^\star$ under ith user's public key, range proofs for $b_i^\star$, and a range proof for the remaining balance $b - \hat{b}$. Thus, there will be a total of $n + 1$ range proofs as opposed to $2n$, a saving of about 50%.

Locking mechanism. When a Zether account is locked to a smart contract, it will only be affected by transactions coming from that contract. Specifically, no one else can deposit or withdraw money from the account. There could be other more flexible ways of locking accounts though. For instance, one could allow locked accounts to be funded from *any* address.

9 Proofs and Σ-Bullets

Transfer and AnonTransfer are medium-sized relations that involve proofs on encrypted data. We, therefore, want to use a proving system which a) is efficient, i.e. has short proofs and efficient verification and b) allows proofs on cryptographically encoded data. Bulletproofs [16] 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 well with Confidential Transactions [52] as it directly proofs of statements which include values that are committed to in a Pedersen commitment. Its short proofs and trustless setup make Bulletproofs an intriguing choice for Zether's underlying proof system. However, unlike UTXO-based Confidential Transactions Zether relies on ElGamal encryptions instead of Pedersen commitments. We, therefore, aim to use a proving system similar to Bulletproofs but where we can prove statements on ElGamal encrypted values. Simply replacing Pedersen commitments with ElGamal encryptions is insufficient as ElGamal encryptions cannot be 'opened' like a commitment and are also not additively homomorphic if encryptions are under different keys, as is the case in Zether. Further for AnonTransfer we need to combine a one-out-of-many proof[4] with range proofs. The one-out of many proof is used to select the receiver and sender transfer encryption and the range proof ensures that no overdraft is created. Bulletproofs enables efficient range proofs and there are logarithmic sized efficient Σ protocols [25] for doing one-out-of-many proofs [13, 41].

To efficiently prove these statements and instantiate Zether we design Σ-Bullets. Σ-Bullets is an extension of Bulletproofs. It enables the combination of Σ-protocols and Bulletproofs. Concretely 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 [21] with the generic circuit-base proof system Bulletproofs. This will benefit other systems that want to prove statements on additively encoded witnesses.

Σ-Bullets. Bulletproofs [61] 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 [14]. The prover then receives challenges y, z and commits to polynomial $t(X)$ using a polynomial

---

[4]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

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. Concretely assume that 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 $\vec{a} \in \mathbb{Z}_p^n, \vec{v} \in \mathbb{Z}_p^m$ such that:

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

Prover		Verifier
Commit to wires values $\vec{a}$ in A		
Commit to randomness in S		
	$\xrightarrow{\quad A, S \quad}$	
		$y, z \leftarrow_\$ \mathbb{Z}_q$
	$\xleftarrow{\quad y, z \quad}$	
Use y to combine constraints		
Such that they can be written as:		
$t(X) + \sum_{i=1}^{n} z^i v_i = \langle l(X), r(X) \rangle \in \mathbb{Z}_q$		
$\wedge\, t(0) = \delta(y, z)$		
$l(X), r(X) \in \mathbb{Z}_q^n \wedge deg(l, r) \leq 3$		
$T = \mathrm{Commit}(t(X))$		
	$\xrightarrow{\quad T \quad}$	
		$x \leftarrow_\$ \mathbb{Z}_q$
	$\xleftarrow{\quad x \quad}$	
$\hat{t} = t(x), \vec{l} = l(x), \vec{r} = r(x)$		
	$\xrightarrow{\quad \hat{t}, \vec{l}, \vec{r} \quad}$	
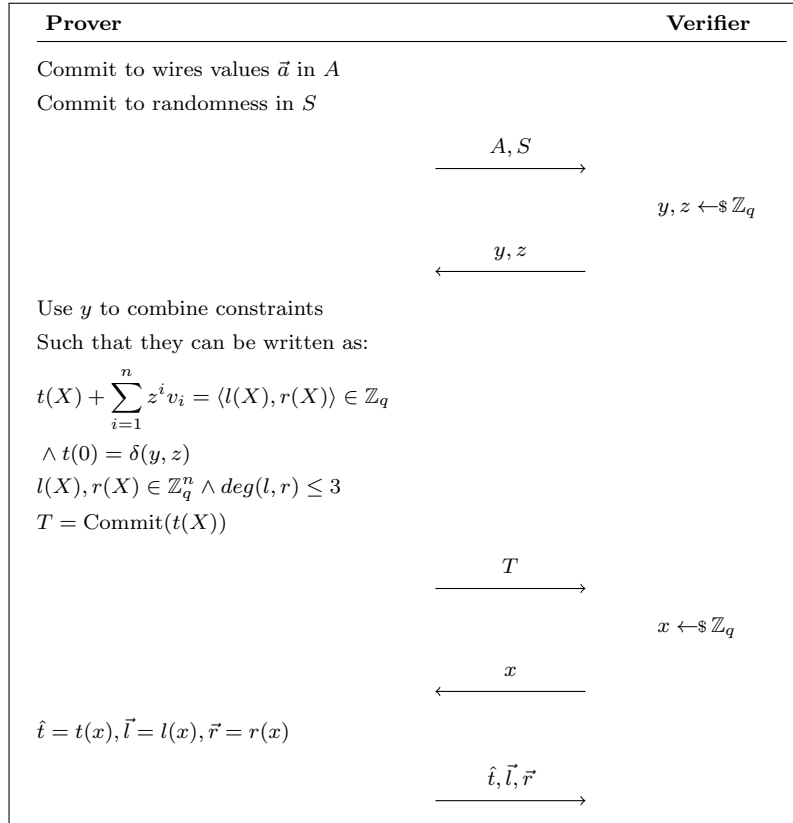
Now the verifier creates commitment P to $l(X), r(X)$ from A, S, y, z and checks the following condition:

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

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
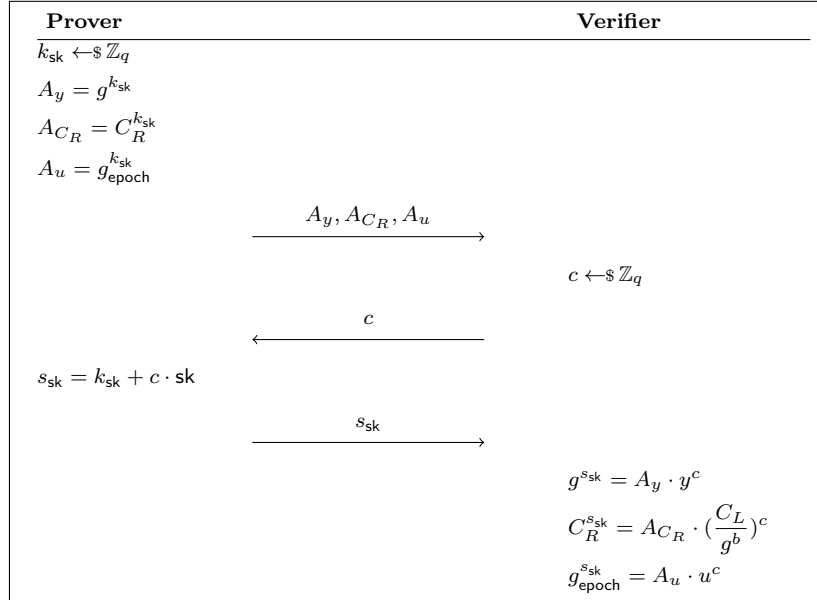
that the prover *proves* that

$$\hat{t} = \sum_{i=1}^{m} v_i \cdot z^i + \delta(y, z) + Open(T).\tag{9}$$

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 (9) 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 [16]. 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.

Burn proof. Σ protocol for proving Statement $\mathsf{st}_{\mathsf{Burn}}$

Prover	**Verifier**
$k_{\mathsf{sk}} \leftarrow\!\!\$\ \mathbb{Z}_q$	
$A_y = g^{k_{\mathsf{sk}}}$	
$A_{C_R} = C_R^{k_{\mathsf{sk}}}$	
$A_u = g_{\mathsf{epoch}}^{k_{\mathsf{sk}}}$	
$\xrightarrow{\quad A_y, A_{C_R}, A_u \quad}$	
	$c \leftarrow\!\!\$\ \mathbb{Z}_q$
$\xleftarrow{\quad c \quad}$	
$s_{\mathsf{sk}} = k_{\mathsf{sk}} + c \cdot \mathsf{sk}$	
$\xrightarrow{\quad s_{\mathsf{sk}} \quad}$	
	$g^{s_{\mathsf{sk}}} = A_y \cdot y^c$
	$C_R^{s_{\mathsf{sk}}} = A_{C_R} \cdot (\frac{C_L}{g^b})^c$
	$g_{\mathsf{epoch}}^{s_{\mathsf{sk}}} = A_u \cdot u^c$

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 $g^{\hat{\mathsf{sk}}} = y \wedge C_R^{\hat{\mathsf{sk}}} = \frac{C_L}{g^b} \wedge g_{\mathsf{epoch}}^{\hat{\mathsf{sk}}} = u$

Σ-Bullets ConfTransfer proof. We will now describe how exactly we use the Σ Bullets construction to create an efficient proof for $\mathsf{st_{Transfer}}$ 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 $\mathsf{st_{ConfTransfer}}$.

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: $(\hat{t}, \tau)$. The protocol 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.

$$\Big\{ (y, \overline{y}, C_{L,n}, C_{R,n}, C, D, \overline{C}, z, \hat{t}, \delta(y,z), \tau; \mathsf{sk}, b^\star, b', r) :$$
$$C = g^{b^\star} y^r \wedge \overline{C} = g^{b^\star} \overline{y}^r \wedge D = g^r \wedge$$
$$C_{L,n} = g^{b'} C_{R,n}^{\mathsf{sk}} \wedge g^{\mathsf{sk}} = y \wedge$$
$$t = \hat{t} - \delta(y,z) \wedge g^{t - b^\star \cdot z^2 - b' \cdot z^3} h^\tau = T_{1,2} \Big\} \quad (10)$$

Prover$(\mathsf{sk}, r, b^\star, b')$	**Verifier**
$k_{\mathsf{sk}}, k_r, k_b \leftarrow\!\!\$\ \mathbb{Z}_q$	
$A_y = g^{k_{\mathsf{sk}}}$	
$A_D = D^{k_r}$	
$A_b = g^{k_b} D^{k_{\mathsf{sk}} z^2} C_{R,n}^{k_{\mathsf{sk}} z^3}$	
$A_{\overline{y}} = (\frac{y}{\overline{y}})^{k_r}$	
$A_t = g^{-k_b}$	

$$\xrightarrow{\ \ A_y, A_D \ \ }$$
$$\xrightarrow{\ \ A_b, A_{\overline{y}}, A_t \ \ }$$
$$\xleftarrow{\quad c \quad} \qquad c \leftarrow\!\!\$\ \mathbb{Z}_q$$

$$s_{\mathsf{sk}} = k_{\mathsf{sk}} + c\mathsf{sk}$$
$$s_r = k_r + cr$$
$$s_b = k_b + c(b^\star z^2 + b' z^3)$$

$$\xleftarrow{\ \ s_{\mathsf{sk}}, s_r, s_b \ \ }$$

Check the following:
$$g^{s_{\mathsf{sk}}} = A_y y^c$$
$$g^{s_r} = A_D D^c$$
$$g^{s_b} = (\frac{C^c}{D^{s_{\mathsf{sk}}}})^{z^2} (\frac{C_{L,n}}{C_{R,n}^{s_{\mathsf{sk}}}})^{z^3}$$
$$(\frac{y}{\overline{y}})^{s_r} = A_{\overline{y}} (\frac{C}{\overline{C}})^c$$
$$g^{t \cdot c - s_b} h^{\tau \cdot c} = A_t T_{1,2}^c$$

Proof Sketch. We now provide a sketch of our Zether-Σ protocol is secure and is a proof of knowledge for Relation (10). 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 $s_{\mathsf{sk}}, s_r, s_b$. He then computes $A_y, A_D, A_b, A_{\hat{y}}$ and A_t according to the verification equations. Note that they will be identically distributed to the honestly generated ones.

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 $c_2, s_{\mathsf{sk},2}, s_{r,2}, s_{b,2}$ be the second transcript. From them he computes $\mathsf{sk} = \frac{s_{\mathsf{sk}} - s_{\mathsf{sk},2}}{c - c_2}, r = \frac{s_r - s_{r,2}}{c - c_2}, b = \frac{s_b - s_{b,2}}{c - c_2}$. 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 $g^{\hat{t}} = g^{b + \delta(y,z)} T_{1,2}$. 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 $(\mathsf{sk}_i, r_i, b_i), i \in \{1, 2\}$. 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:

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

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

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 $(y, \overline{y}, C_{L,n}, C_{R,n}, C, D, \overline{C}, z, \hat{t}, \delta(y,z), \tau; \mathsf{sk}, b^\star, b', r)$ is in Relation (10).

9.1 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 [4, 39]. 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 [2]. 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.

10 Implementation

In order to properly evaluate the Zether protocol and show its feasibility we implemented basic Zether as an Ethereum smart contract. The implementation shows 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.

10.1 Solidity Implementation

We describe our Ethereum implementation of the basic Zether contract. The implementation was written in Solidity and makes use of several observations. Ethereum recently introduced precompiled contracts for elliptic-curve operations on the curve BN-128 [7]. These precompiled contracts reduce the 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 [75].) As we explain in Section 10.3, 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 [16].

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 $n = 1$. This means that instead of sending scalars a and b, the prover sends the two 4 scalar vectors $[a_1, a_2, a_3, a_4]$ and $[b_1, b_2, b_3, b_4]$. 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.

10.2 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 1.8 Gwei per unit of gas [35] and exchange rate of 210 USD per ETH [50]. At the time of writing, a basic Zether transaction costs less than 3 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.

	Gas Cost	in $	EC Cost	|tx|
Burn	384k	0.15	329k	160 bytes
Fund	260k	0.10	41k	64 bytes
Transfer	7188k	2.72	6455k	1472 bytes
Lock	223k	0.08	83k	128 bytes
Unlock	193k	0.07	83k	96 bytes

Table 2: Gas costs for ZSC methods.

10.3 Optimizations and Ethereum Limitations

At the current gas costs, Zether is barely feasible. The cost of a single confidential-only transfer is just below a block's roughly 8 million gas limit [1], i.e., the maximum global gas consumed per block. Ethereum's computation power currently is very limited. A simplified estimate is that at 3 gas 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 [20, 84]. Zether, as a fairly computation-heavy application, would benefit from increased scalability. There are also multiple more specific and smaller changes that would greatly benefit Zether. The majority of the cost of a transaction 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 [8]. 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 [8]. A further EIP reduces the cost of calling a precompiled contract [66] 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 (0.65 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 [75] or Curve25519-ristretto [27]. Another would be supporting multi-exponentiation techniques that can reduce the number of cryptographic operations needed to verify the range proofs [63].

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 [11] or generated from a hash of the proofs.

11 Applications using Zether

Zether's capability to lock funds to other smart contracts can make several common smart contracts more private. Even the basic version of Zether, which is more efficient, can help designers address a number of privacy concerns. We will focus on this version below.

Before going into the applications, it is worth noting that the Zether contract will never transfer funds without first checking an appropriate burn or transfer proof, even if the request comes from another smart contract whose rules do not permit illegal transfers. This design decision ensures that the security of Zether only depends on itself and not on any outside smart contract. Even a maliciously written or insecure smart contract cannot cause Zether to misbehave.

11.1 Sealed-bid Auctions

Auctions are used on Ethereum to sell tokens, resources, etc. [80]. Ethereum Name Service (ENS) [32], for example, resolves human-readable names into machine-readable identifiers, such as Ethereum addresses and IPFS content hashes, and other identifiers. ENS allocates names through a Vickrey auction, a type of sealed-bid auction where the highest bidder wins but only pays the value of the second highest bid. A total of 767,750 of such auctions have been initiated as of September 2018 [36].

ENS auctions consist of three stages. First, a user initiates an auction on a name by bidding on the name. Other people then get three days to place their own bids. Second, everyone reveals their bids over a period of two days. If a bid is not revealed, then the entire amount is forfeited. In the third stage, the highest bidder is awarded the name but only pays the value of the second highest bid.

In the bidding phase, a bidder is supposed to make two API calls. The first call creates a *sealed* bid by hashing the bid value with a bidder generated secret. The second call submits the committed bid along with a certain amount of ether. ENS suggests that a bidder should deposit more ether than the bid value to disguise the true value of the latter.

Drawbacks. One can easily see why this mechanism does not provide good bid privacy. First, the ether deposited reveals an upper bound on the bid value. Second, the bidder may not want to deposit a very large amount of ether (assuming she has it) because, in the best case, the amount remains locked till the end of the bidding phase, and, in the worst case, till the end of the whole bidding process.

Using Zether. Zether provides a simple way to deal with these problems.

- *Bidding phase*: To bid b ETH in the bidding phase, a bidder, say B, can transfer b ZTH from her primary Zether account into a new account, and then lock the latter to the auction contract, call it AUC. Thus, the bid value is completely hidden without B having to deposit any extra money to obfuscate it.

- *Reveal phase*: B reveals her bid by sending b and a burn proof to AUC. AUC can check the proof and verify that b is in fact the right value. ZSC may also export the proof checking functionality as an API call. Now, there are three possibilities:

  1. If a bid is not revealed in this phase or the revealed value is incorrect, then AUC does not do anything. Bidder's account remains locked. Thus, bidders have a strong incentive to reveal their bid regardless of the outcome.
  2. If a new revealed bid is lower, then AUC can simply unlock the bidder's account.
  3. If a new revealed bid is highest so far, then AUC unlocks the previous highest bid and stores only its amount.

- *Final phase*: For the winning bid, AUC must only keep the next highest bid, say $b^\star$, and release the difference back to the bidder. This can be done through a Transfer transaction. However, since all amounts are public anyway, AUC can also simply burn the whole amount and retain one part (the winner's payment) while refunding the rest to the winner. This prevents AUC from having to create an expensive Transfer transaction and it can utilize the burn proof which was submitted with the bid. For simplicity, ZSC can provide a functionality to split up the burn refund directly.

11.2   Payment Channels

Payment channels are a popular way to avoid the fee and delay associated with on-chain transactions. Two or more parties can set-up a payment channel amongst them by committing to some money up front. They can then send messages to each other offline to keep track of the money they owe each other.

The first payment channels were proposed for Bitcoin [29, 65, 78]. Simpler constructions have been developed for Ethereum [5, 69]. We will consider a very simple construction here given by Di Ferrante [38] to illustrate some of the confidentiality issues surrounding payment channels and how Zether can help alleviate them. The ideas described below would extend easily to more complex channels.

Suppose Bob wants to make small payments to Alice every time she *tweets* for him. Bob can open a payment channel with Alice with the help of a smart contract, say PC. Bob deposits a certain amount of ETH with PC and sets an expiration date. Whenever Alice tweets, Bob signs the *total* amount of money he owes to Alice so far, and sends the signature to Alice. At any time, Alice can cash out by sending the latest signature to the contract. PC will pay Alice accordingly and send the remaining balance to Bob. (We have omitted several details here because they are not important for the rest of the discussion; please refer to original write-up [38] for them.)

Drawbacks. The amount of ETH Bob deposits with PC is visible to everyone. Indeed, even Alice does not need to know this amount as long as she is convinced that PC holds enough money to pay her off. Further, when Alice cashes out, everyone can see how much she charges per tweet (since the number of tweets is also public).

Using Zether. Zether can easily be used to create a confidential payment channel. Instead of depositing ETH directly with PC, Bob will lock his Zether account to PC (or, as in the previous example, transfer some money to a new Zether account and lock that to PC). Whenever Alice tweets, Bob would send a Zether transfer transaction to Alice of the amount that she expects, say b. Alice will check the validity of the transaction and that it indeed transfers b to her.

When Alice wants to cash out, she would send the latest transaction to PC, who will pass it on ZSC. ZSC will process the transfer because it is locked to PC. A transfer also unlocks Bob's account. If Alice never cashes out, PC will just unlock Bob's account after an expiration date. As a result, the payment channel between Alice and Bob remains confidential to others. Furthermore, Alice does not learn the total amount Bob locked into the payment channel.

Using a simple signed counter the proposal can be trivially extended to bi-directional channels which in turn can be used to build payment channel networks [30,65]. With a slightly more involved design we can also enable adding more funds to a channel. Suppose Bob wants to add more ZTH to his account. He prepares a transaction for PC and then lets Alice sign the transaction. He then forwards the transaction and the signature to PC which checks both and, if valid, submits the transaction to ZSC. Note that Alice needs to sign the transaction because it might invalidate a previous payment channel transfer. Alice will require that Bob give him a new valid Zether transaction with respect to Bob's new balance.

Green and Miers [40] recently proposed new techniques to construct anonymous payment channels. Their design is more involved but provides strong privacy against an intermediary payment channel hub. The design requires an underlying private currency and as such is compatible with Zether.

11.3 Stake Voting

Any voting process must be as open and transparent as possible. A blockchain based solution can provide such benefits but voter privacy becomes a concern. McCorry et al. [54] proposed a decentralized mechanism for boardroom voting, called the Open Vote Network (OVN), which provides voter privacy. They implement OVN as a Ethereum smart contract.

The Election is carried out in several stages. In the voting stage, a participant publishes a special encryption of their vote v_i and a ZK-proof that v_i is binary. In the tally stage, the votes can be summed up to compute the final tally. The encryption scheme is designed such that that only the final tally can be computed—individual votes remain private.

A natural question is whether votes can carry weights, instead of just being zero or one. For instance, can participants cast votes of size proportional to their assets, without revealing the value of those assets? Zether provides a simple way to achieve this. In the sign-up phase of the election, participants will lock their Zether account to the voting smart contract. Then, during the voting stage, when a participant casts a vote v_i (in an encrypted form), he/she will provide a ZK-proof that v_i is equal to the (encrypted) amount in his/her locked account.

11.4 Privacy-preserving Proof-of-Stake

Proof-of-stake (PoS) is a consensus mechanism for blockchains, where a participant is elected as a leader with a probability that is proportional to the amount of coins that the participant *stakes* in the election. There are several PoS protocols [23,28,45] which all roughly follow the following simplified design: Users *stake* a number of coins and then a random beacon [9,11,68] is used to select one or a subset of them as leaders. Ethereum is planning to switch to a PoS-

based consensus model in the future [20]. All current PoS proposals are for non-confidential currencies and require the user to reveal its stake.

We propose a mechanism to make the PoS election process confidential using Zether, i.e., users do not have to reveal their stake in order to participate in the election. Similar ideas could be used to implement PoS for other confidential currencies such as ZCash [3] or Monero [82]. The general idea is that users 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, i.e. the PoS election. The user can use a range proof to prove that v is in fact between t and $t + b$ without revealing t or b. This range proof is similar to the one used in basic Zether's Transfer.

It is further possible to adjust the winning probability by scaling the range appropriately. Let p denote the desired probability of winning for a single unit of stake. Let q be the size of the ring in which the ElGamal encrypted values lie. For simplicity assume that $p \cdot q$ is an integer. A users wins if v falls in between t and $t + b \cdot p \cdot q$. This happens with probability $b \cdot p$.

The hidden lottery approach does not guarantee that at least one user wins. In order to guarantee this, we can have a series of lottery drawings by utilizing a hash function $\mathcal{H}$ that is modeled as a random oracle. For beacon v, we define $v_i = \mathcal{H}(v||i)$ as a sequence of lottery drawings. The user who can demonstrate a winning ticket on the lowest drawing wins. We can also ensure that exactly 1 participant wins. First, we ensure that the lottery drawings are uncorrelated by appending the serialized public key pk when computing the lottery drawings, i.e. $v_{i,\mathsf{pk}} = \mathcal{H}(v||i||\mathsf{pk})$. Even a malicious user will not be able to correlate lottery tickets for distinct public keys, if $\mathcal{H}$ is in fact a random function. Then we use run-off lottery elections between the previous winners until a single winner remains.

The proposed scheme has the severe flaw that after multiple drawings an adversary will be estimate a user's stake by simply observing the user's probability of winning. This can be circumvented by having a user *prove* in zero-knowledge that she knows a winning ticket instead of revealing it. We can use a similar ZK-proof as the anonymous Zether transfer. The user proves that she *knows* a private key which won the lottery instead of revealing her stake. We can use a similar nonce protection as for the anonymous transfer to prevent that a user claims the same winning lottery ticket twice.

The proposed scheme requires a formal analysis and further investigation which we leave for future work.

12   Conclusion

Blockchain-based, smart contract platforms have great promise to remove trust and add transparency to distributed applications. Currently, this benefit often comes at the cost of greatly reduced privacy. With Zether, we begin to bridge the gap, showing that smart contracts can keep their accounts private while not losing functionality and with only a limited performance overhead. We do this by building a confidential and anonymous token on top of Ethereum. Zether uses account-based private payments which are of independent interest as many popular ledger systems currently use an account-based but non-private data structure. We show the versatility of our work by showing multiple applications that can be built using Zether. In order to efficiently instantiate Zether, we extend Bulletproofs to efficiently interact with Σ protocols.

References

[1] Ethereum network status. https://ethstats.net/, 2018. (Accessed on 12/03/2018).

[2] M. Abdalla, J. H. An, M. Bellare, and C. Namprempre. From identification to signatures via the Fiat-Shamir transform: Minimizing assumptions for security and forward-security. In L. R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 418–433. Springer, Heidelberg, Apr. / May 2002.

[3] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society Press, May 2014.

[4] E. Ben-Sasson, A. Chiesa, and N. Spooner. Interactive oracle proofs. In M. Hirt and A. D. Smith, editors, *TCC 2016-B, Part II*, volume 9986 of *LNCS*, pages 31–60. Springer, Heidelberg, Oct. / Nov. 2016.

[5] I. Bentov, R. Kumaresan, and A. Miller. Instantaneous decentralized poker. Cryptology ePrint Archive, Report 2017/875, 2017. http://eprint.iacr.org/2017/875.

[6] A. Biryukov, D. Khovratovich, and I. Pustogarov. Deanonymisation of clients in bitcoin P2P network. In G.-J. Ahn, M. Yung, and N. Li, editors, *ACM CCS 14*, pages 15–29. ACM Press, Nov. 2014.

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

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

[9] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. Verifiable delay functions. In H. Shacham and A. Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 757–788. Springer, Heidelberg, Aug. 2018.

[10] D. Boneh and V. Shoup. *A graduate course in applied cryptography.* Cambridge, 2018.

[11] J. Bonneau, J. Clark, and S. Goldfeder. On bitcoin as a public randomness source. Cryptology ePrint Archive, Report 2015/1015, 2015. http://eprint.iacr.org/2015/1015.

[12] J. Bonneau, A. Narayanan, A. Miller, J. Clark, J. A. Kroll, and E. W. Felten. Mixcoin: Anonymity for bitcoin with accountable mixes. In N. Christin and R. Safavi-Naini, editors, *FC 2014*, volume 8437 of *LNCS*, pages 486–504. Springer, Heidelberg, Mar. 2014.

[13] J. Bootle, A. Cerulli, P. Chaidos, E. Ghadafi, J. Groth, and C. Petit. Short accountable ring signatures based on DDH. In G. Pernul, P. Y. A. Ryan, and E. R. Weippl, editors, *ESORICS 2015, Part I*, volume 9326 of *LNCS*, pages 243–265. Springer, Heidelberg, Sept. 2015.

[14] J. Bootle, A. Cerulli, P. Chaidos, J. Groth, and C. Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In M. Fischlin and J.-S. Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 327–357. Springer, Heidelberg, May 2016.

[15] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu. Zexe: Enabling decentralized private computation. Cryptology ePrint Archive, Report 2018/962, 2018. https://eprint.iacr.org/2018/962.

[16] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. 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.

[17] V. Buterin. Ethereum's white paper. https://github.com/ethereum/wiki/wiki/White-Paper, 2014.

[18] V. Buterin. On abstraction. https://blog.ethereum.org/2015/07/05/on-abstraction, 2015.

[19] V. Buterin. Thoughts on UTXOs. https://medium.com/@ConsenSys/thoughts-on-utxo-by-vitalik-buterin-2bb782c67e53, 2016.

[20] V. Buterin and V. Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.

[21] J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In M. Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 61–76. Springer, Heidelberg, Aug. 2002.

[22] CHAINALYSIS. https://www.chainalysis.com.

[23] J. Chen, S. Gorbunov, S. Micali, and G. Vlachos. ALGORAND AGREEMENT: Super fast and partition resilient byzantine agreement. Cryptology ePrint Archive, Report 2018/377, 2018. https://eprint.iacr.org/2018/377.

[24] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. M. Johnson, A. Juels, A. Miller, and D. Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contract execution. *CoRR*, abs/1804.05141, 2018.

[25] R. Cramer and I. Damgård. Zero-knowledge proofs for finite field arithmetic; or: Can zero-knowledge be for free? In H. Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 424–441. Springer, Heidelberg, Aug. 1998.

[26] R. Cramer, R. Gennaro, and B. Schoenmakers. A secure and optimally efficient multi-authority election scheme. In W. Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 103–118. Springer, Heidelberg, May 1997.

[27] Curve25519-ristretto. https://ristretto.group/.

[28] B. David, P. Gazi, A. Kiayias, and A. Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In J. B. Nielsen and V. Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 66–98. Springer, Heidelberg, Apr. / May 2018.

[29] C. Decker and R. Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In A. Pelc and A. A. Schwarzmann, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 3–18. Springer International Publishing, 2015.

[30] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski. PERUN: Virtual payment channels over cryptographic currencies. Cryptology ePrint Archive, Report 2017/635, 2017. http://eprint.iacr.org/2017/635.

[31] ELLIPTIC. https://www.elliptic.co.

[32] Ethereum Name Service. https://ens.domains.

[33] EOS.io — Blockchain software architecture. https://www.eos.io.

[34] Ethereum – Create a Crowdsale. https://www.ethereum.org/crowdsale.

[35] Ethereum Gasstation. https://ethgasstation.info/calculatorTxV.php.

[36] Etherscan – Ether Name Sservice. https://etherscan.io/ens.

[37] I. Eyal. The miner's dilemma. In *2015 IEEE Symposium on Security and Privacy*, pages 89–103. IEEE Computer Society Press, May 2015.

[38] M. D. Ferrante. Ethereum Payment Channel in 50 Lines of Code, 2017. https://medium.com/@matthewdif/ethereum-payment-channel-in-50-lines-of-code-a94fad2704bc.

[39] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In A. M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, Aug. 1987.

[40] M. Green and I. Miers. Bolt: Anonymous payment channels for decentralized currencies. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *ACM CCS 17*, pages 473–489. ACM Press, Oct. / Nov. 2017.

[41] J. Groth and M. Kohlweiss. One-out-of-many proofs: Or how to leak a secret and spend a coin. In E. Oswald and M. Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 253–280. Springer, Heidelberg, Apr. 2015.

[42] E. Heilman, L. Alshenibr, F. Baldimtsi, A. Scafuro, and S. Goldberg. TumbleBit: An untrusted bitcoin-compatible anonymous payment hub. In *NDSS 2017*. The Internet Society, Feb. / Mar. 2017.

[43] E. Heilman, F. Baldimtsi, and S. Goldberg. Blindly signed contracts: Anonymous on-blockchain and off-blockchain bitcoin transactions. In J. Clark, S. Meiklejohn, P. Y. A. Ryan, D. S. Wallach, M. Brenner, and K. Rohloff, editors, *FC 2016 Workshops*, volume 9604 of *LNCS*, pages 43–60. Springer, Heidelberg, Feb. 2016.

[44] G. Kappos, H. Yousaf, M. Maller, and S. Meiklejohn. An Empirical Analysis of Anonymity in Zcash. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 463–477, 2018.

[45] A. Kiayias, A. Russell, B. David, and R. Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In J. Katz and H. Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 357–388. Springer, Heidelberg, Aug. 2017.

[46] A. E. Kosba, A. Miller, E. Shi, Z. Wen, and C. 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.

[47] A. Kumar, C. Fischer, S. Tople, and P. Saxena. A traceability analysis of monero's blockchain. In S. N. Foley, D. Gollmann, and E. Snekkenes, editors, *ESORICS 2017, Part II*, volume 10493 of *LNCS*, pages 153–173. Springer, Heidelberg, Sept. 2017.

[48] K. Kurosawa. Multi-recipient public-key encryption with shortened ciphertext. In D. Naccache and P. Paillier, editors, *PKC 2002*, volume 2274 of *LNCS*, pages 48–63. Springer, Heidelberg, Feb. 2002.

[49] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, New York, NY, USA, 2016. ACM.

[50] Total Market Capitalization. https://coinmarketcap.com/charts.

[51] G. Maxwell. Coinjoin: Bitcoin privacy for the real world, 2013. https://bitcointalk.org/?topic=279249.

[52] G. Maxwell. Confidential transactions, 2015. https://people.xiph.org/~greg/confidential_values.txt.

[53] D. Mazieres. The stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation*, 2015.

[54] P. McCorry, S. F. Shahandashti, and F. Hao. A smart contract for boardroom voting with maximum voter privacy. In A. Kiayias, editor, *FC 2017*, volume 10322 of *LNCS*, pages 357–375. Springer, Heidelberg, Apr. 2017.

[55] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage. A fistful of bitcoins: Characterizing payments among men with no names. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 127–140, New York, NY, USA, 2013. ACM.

[56] I. Miers, C. Garman, M. Green, and A. 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.

[57] A. Miller and I. Bentov. Zero-collateral lotteries in bitcoin and ethereum. In *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, pages 4–13, April 2017.

[58] M. Möser, K. Soska, E. Heilman, K. Lee, H. Heffan, S. Srivastava, K. Hogan, J. Hennessey, A. Miller, A. Narayanan, and N. Christin. An empirical analysis of traceability in the monero blockchain. *PoPETs*, 2018(3):143–163, 2018.

[59] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *bitcoin.org*, 2009.

[60] S. Noether. Ring signature confidential transactions for monero. Cryptology ePrint Archive, Report 2015/1098, 2015. http://eprint.iacr.org/2015/1098.

[61] B. B. nz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. Cryptology ePrint Archive, Report 2017/1066, 2017. https://eprint.iacr.org/2017/1066.

[62] T. 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, London, UK, UK, 1992. Springer-Verlag.

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

[64] A. Poelstra. Mimblewimble, 2016. https://scalingbitcoin.org/papers/mimblewimble.pdf.

[65] J. Poon and T. Dryja. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments, 2016. https://lightning.network/lightning-network-paper.pdf.

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

[67] J. Quesnelle. On the linkability of Zcash transactions. *CoRR*, abs/1712.01210, 2017.

[68] M. O. Rabin. Transaction protection by beacons. *Journal of Computer and System Sciences*, 27(2):256–267, 1983.

[69] The Raiden Network. https://raiden.network.

[70] Ripple - One Frictionless Experience To Send Money Globally. https://www.ripple.com.

[71] D. Ron and A. Shamir. Quantitative analysis of the full Bitcoin transaction graph. In A.-R. Sadeghi, editor, *FC 2013*, volume 7859 of *LNCS*, pages 6–24. Springer, Heidelberg, Apr. 2013.

[72] J. Rubin. The collapse of ETH is inevitable. https://techcrunch.com/2018/09/02/the-collapse-of-eth-is-inevitable, 2018.

[73] T. Ruffing, P. Moreno-Sanchez, and A. Kate. CoinShuffle: Practical decentralized coin mixing for bitcoin. In M. Kutylowski and J. Vaidya, editors, *ESORICS 2014, Part II*, volume 8713 of *LNCS*, pages 345–364. Springer, Heidelberg, Sept. 2014.

[74] C.-P. Schnorr. Efficient identification and signatures for smart cards (abstract) (rump session). In J.-J. Quisquater and J. Vandewalle, editors, *EUROCRYPT'89*, volume 434 of *LNCS*, pages 688–689. Springer, Heidelberg, Apr. 1990.

[75] Secp256k1. https://en.bitcoin.it/wiki/Secp256k1.

[76] Sequence. https://chain.com/sequence.

[77] Solidity webpage. https://solidity.readthedocs.io, 2017.

[78] J. Spilman. Anti DoS for tx replacement, 2013. https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2013-April/002433.html.

[79] N. Szabo. Smart contracts: Building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought*, 16, 1996.

[80] Analyzing Token Sale Models. https://vitalik.ca/general/2017/06/09/sales.html.

[81] Tor Project. https://www.torproject.org.

[82] N. van Saberhagen. CryptoNote v 2.0, 2013. https://cryptonote.org/whitepaper.pdf.

[83] G. Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. https://gavwood.com/paper.pdf.

[84] V. Zamfir. Casper the friendly ghost: A correct by construction blockchain consensus protocol. https://github.com/ethereum/research/blob/master/papers/CasperTFG/CasperTFG.pdf, 2017.

A   Correctness

To formally state the correctness property, we define what it means for a transaction to be generated honestly and the notion of an ideal state.

Honestly-generated transactions.   Let $\mathsf{TX} = (\mathsf{TX}_1, \mathsf{TX}_2, \ldots, \mathsf{TX}_m)$ be a group of transactions such that for every $i \in [m]$, $\mathsf{TX}_i = (\mathsf{tx}_{i,1}, \mathsf{tx}_{i,2}, \ldots)$ is a sequence of transactions which are processed in that order into the block at height h_i ($h_1 < h_2 < \ldots < h_m$). Define the k-th transaction in TX to be the k-th transaction in the sequence $(\mathsf{tx}_{1,1}, \mathsf{tx}_{1,2}, \ldots, \mathsf{tx}_{2,1}, \mathsf{tx}_{2,2}, \ldots, \ldots, \mathsf{tx}_{m,1}, \mathsf{tx}_{m,2}, \ldots)$. We say that these transactions were generated honestly if all of the following are true:

- All secret key, address pairs are generated through the CreateAddress algorithm.

- For all i, j, $\mathsf{tx}_{i,j}$ is generated by running one of the five user algorithms for generating transactions.

- For any $\mathsf{TX}_{i_1}, \mathsf{TX}_{i_2}, \ldots$ s.t. $h_{i_1}, h_{i_2}, \ldots$ are in the same epoch, and any pk, TX_i contains at most one transfer or burn transaction (not one of each) initiated from pk.

- For any $\mathsf{tx}_{i,j}$, if h_i is in epoch e then the state after the last block of epoch $e-1$ was used to generate $\mathsf{tx}_{i,j}$ (if a state is needed at all).

Recall that honest users are supposed to publish transfer, burn and lock transactions only at the beginning of an epoch. Thus, the last point above implies that even if there is some delay in processing the transaction, it is no more than the epoch length. In other words, all transactions generated for a certain epoch are assumed to be processed in the same epoch.

For a block height h, let $\mathsf{last}(h)$ denote the height of the last block in the previous epoch, i.e. $\mathsf{last}(h) = \lfloor h/\mathsf{E} \rfloor \cdot \mathsf{E} - 1$, and $\mathsf{next}(h)$ denote the height of the first block in the next epoch, i.e. $\mathsf{next}(h) = \lceil (h+1)/\mathsf{E} \rceil \cdot \mathsf{E}$.

Ideal state.   The ideal state IdSt consists of two tables Bal and Lk indexed by addresses. For a public key pk, $\mathsf{Bal}[\mathsf{pk}]$ gives the balance in pk and $\mathsf{Lk}[\mathsf{pk}]$ gives the locking Ethereum address (or $\bot$ if pk is not locked). Initially, $\mathsf{Bal}[\mathsf{pk}]$ and $\mathsf{Lk}[\mathsf{pk}]$ are set to 0 and $\bot$, respectively,

for all pk.[5] These tables are updated as each new transaction is processed. We use Bal_h and Lk_h to denote the state of these tables when all the transactions included in blocks $0, 1, \ldots, h$ have been processed.

Ideal state for a set of transactions TX is defined recursively. Let $\mathsf{okToOperate}$ be a function that given a public key pk and an Ethereum address addr, returns 1 iff $\mathsf{Lk}[\mathsf{pk}]$ is addr or $\perp$. (In either case, addr is allowed to operate on the account.) Let IdSt_k denote the ideal state after the first k transactions have been processed. IdSt_0, ideal state at the beginning, is set to be the initial state of Bal and Lk tables. IdSt_k is defined based on IdSt_{k-1} and the kth transaction as follows. If the kth transaction $\mathsf{tx}_{i,j}$ is:

- $\mathsf{tx}_{\mathsf{fund}} \leftarrow \mathsf{CreateFundTx}(\mathsf{pk}, \mathsf{amt})$: if the sum of all balances and amt is at most MAX, and $\mathsf{okToOperate}(\mathsf{pk}, \mathsf{tx}_{\mathsf{fund}}.\mathsf{addr}) = 1$, then $\mathsf{Bal}[\mathsf{pk}]$ is incremented by amt.

- $\mathsf{tx}_{\mathsf{trans}} \leftarrow \mathsf{CreateTransferTx}(\mathsf{sk}_{\mathsf{from}}, \mathsf{pk}_{\mathsf{to}}, \mathsf{AnonSet}, \mathsf{amt}, \mathsf{st}_{\mathsf{last}(h_i)})$: if

  - for every $\mathsf{pk} \in \mathsf{AnonSet}$, $\mathsf{okToOperate}(\mathsf{pk}, \mathsf{tx}_{\mathsf{trans}}.\mathsf{addr}) = 1$, and
  - $\mathsf{Bal}_{\mathsf{last}(h_i)}[\mathsf{pkOf}(\mathsf{sk}_{\mathsf{from}})] \geq \mathsf{amt}$,

  then $\mathsf{Bal}[\mathsf{pkOf}(\mathsf{sk}_{\mathsf{from}})]$ is decremented by amt and $\mathsf{Bal}[\mathsf{pk}_{\mathsf{to}}]$ is incremented by amt.

- $\mathsf{tx}_{\mathsf{lock}} \leftarrow \mathsf{CreateLockTx}(\mathsf{sk}, \mathsf{addr})$: if $\mathsf{okToOperate}\ (\mathsf{pk}, \mathsf{tx}_{\mathsf{lock}}.\mathsf{addr}) = 1$ then $\mathsf{Lk}_{\mathsf{next}(h)}[\mathsf{pk}] = \mathsf{addr}$.

- $\mathsf{tx}_{\mathsf{unlock}} \leftarrow \mathsf{CreateUnlockTx}(\mathsf{pk})$: if $\mathsf{okToOperate}\ (\mathsf{pk}, \mathsf{tx}_{\mathsf{unlock}}.\mathsf{addr}) = 1$ then $\mathsf{Lk}_{\mathsf{next}(h)}[\mathsf{pk}] = \perp$.

- $\mathsf{tx}_{\mathsf{burn}} \leftarrow \mathsf{CreateBurnTx}(\mathsf{sk}, \mathsf{amt}, \mathsf{st}_{\mathsf{last}(h_i)})$: if $\mathsf{okToOperate}\ (\mathsf{pk}, \mathsf{tx}_{\mathsf{burn}}.\mathsf{addr}) = 1$ and $\mathsf{Bal}_{\mathsf{last}(h_i)}[\mathsf{pk}] \geq \mathsf{amt}$, then decrement $\mathsf{Bal}[\mathsf{pk}]$ by amt.

(If in any of the above cases, pk is not defined, then it must be taken to mean $\mathsf{pkOf}(\mathsf{sk})$.)

Correctness. A payment mechanism is **correct** if for every group of transactions TX, whenever $\mathsf{tx}_{i,j}$ is a burn transaction $\mathsf{tx}_{\mathsf{burn}} = (\mathsf{pk}, \mathsf{amt}, \ldots)$, $\mathsf{Burn}(\mathsf{tx}_{\mathsf{burn}}).\mathsf{ETH}$ is same as $\mathsf{Bal}_{\mathsf{last}(h_i)}[\mathsf{pk}]$. (Recall that $\mathsf{Burn}(\mathsf{tx}_{\mathsf{burn}}).\mathsf{ETH}$ is the amount of Ether sent back to the user who called Burn with $\mathsf{tx}_{\mathsf{burn}}$.) Furthermore, for every sk and h, we require that $\mathsf{ReadBalance}(\mathsf{sk}, \mathsf{st}_h)$ returns $\mathsf{Bal}_{\mathsf{last}(h)}[\mathsf{pkOf}(\mathsf{sk})]$.

B Security Requirements

We define here the common elements of the two security games; we refer to this basic version as just $\mathsf{Security\text{-}Game}$. All the entities ($\mathsf{Chal}$, Adv and $\mathcal{O}_{\mathsf{SC}}$) in the game receive the security parameter λ as input. When $\mathcal{O}_{\mathsf{SC}}$ receives a transaction from Chal or Adv, it adds the transaction to the queue of pending transactions. At any time, Chal can get the latest state of SC (or any older state) through $\mathcal{O}_{\mathsf{SC}}$. 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.

Adversary's influence on the contract. We allow Adv to control the state of SC in one of the following ways:

---

[5]A more *natural* way to define ideal state would be to say that addresses are added as they are used in transactions. The formulation here, however, is simpler and does not lose any generality.

1. It can instruct Chal to run one of the user algorithms with certain inputs and send the resulting transaction (if any) to $\mathcal{O}_{\mathsf{SC}}$ from an Ethereum address of its choice . Example instructions are (addr, CreateAddress), and (addr, CreateFundTx, pk, amt). Note that:

   - For a CreateAddress instruction, Chal sends the address output by this algorithm—but not the secret key—directly to Adv.

   - For transactions that require a secret key under the possession of Chal as an input, Adv specifies the corresponding public key instead.

   - When Chal receives an instruction, it uses the state of SC in the last block of the previous epoch (if needed) to execute the instruction.

   - If Chal has generated a transfer or a burn transaction in an epoch with a certain public key, then instructions to generate any one of these two transactions again from the same public key in the same epoch will be rejected. Chal will also reject transfer instructions that require it to use accounts locked to different addresses in the anonymity set. (This point and the one before are important because Chal models honest parties in the system.)

2. It can directly send an arbitrary transaction to $\mathcal{O}_{\mathsf{SC}}$.

3. It can ask $\mathcal{O}_{\mathsf{SC}}$ to process an arbitrary (non-empty) subset of the pending transactions and update the state of SC. This captures the addition of a new block to the blockchain.

The game stops when Adv so desires. Let $h^\star$ be the height of the last block. Let PK denote the set of public keys generated by Chal (at the behest of Adv). Adv does not have the secret keys for them.

B.1 Overdraft-safety

We capture overdraft-safety by means of a game called Overdraft-Safety-Game between a challenger Chal, an adversary Adv and the oracle $\mathcal{O}_{\mathsf{SC}}$, who interact in the exact same way as Security-Game. Let us associate some quantities with the game with respect to Adv:

- EtherDeposited. Denotes the sum of all $\mathsf{tx}_{\mathsf{fund}}.\mathsf{ETH}$ for which $\mathsf{tx}_{\mathsf{fund}} = (\mathsf{pk}, \dots)$ is such that $\mathsf{pk} \notin \mathsf{PK}$.

- ZetherReceived. Let in be the sum of all $\mathsf{tx}_{\mathsf{fund}}.\mathsf{ETH}$ for which $\mathsf{tx}_{\mathsf{fund}} = (\mathsf{pk}, \dots)$ is such that $\mathsf{pk} \in \mathsf{PK}$. Let $\mathsf{honBal} = \sum_{\mathsf{pkOf(sk)} \in \mathsf{PK}} \mathsf{ReadBalance}(\mathsf{sk}, \mathsf{st}_{h^\star})$ be the total balance of honest parties. (Recall that $h^\star$ is the height of the last added block before the game ends.) Finally, let out be the sum of all $\mathsf{burn}(\mathsf{tx}_{\mathsf{burn}}).\mathsf{ETH}$ for which $\mathsf{tx}_{\mathsf{burn}} = (\mathsf{pk}, \dots)$ is such that $\mathsf{pk} \in \mathsf{PK}$. Then ZetherReceived denotes $\mathsf{in} - \mathsf{out} - \mathsf{honBal}$, which may be negative.

- EtherBurnt. Denotes the sum of all $\mathsf{burn}(\mathsf{tx}_{\mathsf{burn}}).\mathsf{ETH}$ for which $\mathsf{tx}_{\mathsf{burn}} = (\mathsf{pk}, \dots)$ is such that $\mathsf{pk} \notin \mathsf{PK}$.

A payment mechanism is *safe against overdrafts* if for all PPT adversaries Adv, the probability that EtherDeposited + ZetherReceived < EtherBurnt in Overdraft-Safety-Game is $\mathsf{negl}(\lambda)$, where the probability is taken over the coin tosses of Adv and the challenger.

B.2 Privacy

In order to capture the privacy requirement formally, we define a game called Privacy-Game between a challenger Chal, an adversary Adv and the oracle $\mathcal{O}_{\mathsf{SC}}$, who interact in the same way as Security-Game but with one crucial difference. Instead of sending just one instruction to

Chal every time, Adv sends two *consistent* instructions at some point and only one instruction rest of the time (consistency will be defined below). Chal executes the $(b+1)$-th instruction based on a bit b hidden from Adv, which is chosen randomly in advance. Adv outputs a bit b' at the end of the game as a guess for b. (This is the typical left-or-right setting used for indistinguishability-based definitions.)

Two instructions are *consistent* if they refer to the same user algorithm and need to be sent from the same Ethereum address. Additionally, if the instructions are for

- creating a fund transaction, then they are associated with the same public key and amount.

- creating a transfer transaction, then

  - they are associated with the same anonymity set;
  - the amounts should be at most the value returned by ReadBalance with the respective sender's public key and the state of the last block in the previous epoch;
  - if any of the receivers is a corrupt party, both instructions have the same receiver and the same amount.

- creating a lock transaction, then they are associated with the same public key and Ethereum address.

- creating an unlock transaction, then they are associated with the same public key.

- creating a burn transaction, then they are associated with the same public key, and ReadBalance with that public key and the state of the last block in the previous epoch returns the same value.

- executing ReadBalance, then they must return the same value.

A payment mechanism is *private* if for all PPT adversaries Adv, the probability that $b' = b$ in Privacy-Game is $1/2 + \mathsf{negl}(\lambda)$, where the probability is over the coin tosses of Adv and challenger.

C   Security Proof

In this section we provide a sketch of a security proof of Zether with respect to the correctness and security definitions from Appendix A and B. (Basic Zether can be proved secure along similar lines.)

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.

An honest user uses the state of the last block in the previous epoch to generate transactions. CreateTransferTx and CreateBurnTx roll over all the input public keys based on this state. 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 as per Appendix B.1.

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 $(y_1, \ldots, y_n)$ be the anonymity set, (C_1, D), $\ldots$, (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 $b_1, \ldots, b_n$ 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 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.

Now we can 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.