Anonymous Zether: Infrastructure

Benjamin E. Diamond
J.P. Morgan

Abstract

We describe supplemental utilities for enterprise deployments of Anonymous Zether.

1 Provably revealing oneself as the sender of a transaction

In Anonymous Zether [Dia19], *even the recipient* of a payment generally possesses no means by which to identify the transaction's sender. In certain settings, this may be desirable; in other settings, a sender may wish to selectively reveal her identity to the recipient (so as, for example, to claim the settlement of an outstanding debt). We specify a procedure through which this voluntary self-identification can be performed. (Actually, the party to whom self-identification is performed can be arbitrary; we assume in what follows that it's the recepient.)

After the transaction is mined, the sender should issue a *private* message to the recipient, which includes the transfer's transaction hash, her own public key y_i, and finally the randomness r used in the ElGamal ciphertexts $(C_i, D)_{i=1}^N$ (see [BAZB, (8)]). (This r should be considered a secret "viewing key" for the transfer.) Upon receiving this message, the recipient should retrieve and parse the relevant transaction payload, use r to decrypt the ciphertext (C_i, D) corresponding to y_i, and finally confirm that this ciphertext's message is nonzero (and in particular that it equals the opposite of the received amount).

This procedure may optionally be embedded at the *Dapp* level, so that senders of transactions are automatically tracked and credited.

We note that the sender must trust the recipient; a leak of the viewing key r would irrevocably unmask the identity of the sender (and of the recipient) to the public. Of interest for future work could be a *deniable* self-identification scheme, convincing only to the recipient (and not to others).

2 Secure issuance and destruction

While the "main-net approach" to Zether—described, for example, in [BAZB]—specifies that each Zether Smart Contract interoperate with a fixed, pre-specified ERC-20-compliant contract, we propose instead that Zether Smart Contracts (ZSCs) operate *standalone*.

Each ZSC should represent (a virtual tokenization of) some particular asset; prior to its deployment, some particular party, or group of parties, must be designed as the official "issuing agent" of the asset. Having generated herself a Zether (i.e., `altbn128`) keypair, this issuing agent should deploy the ZSC in such a way that its constructor pre-populates her account with an encrypted balance containing the full capacity of the contract (for example, $2^{32} - 1$). Her account should be considered the "vault" account.

In order to privately issue tokens to some particular payee, the agent should initiate a transfer from the vault account to this payee. In order to privately destroy assets, a user should simply transfer them back into the vault (and, perhaps, notify the agent using the above procedure, so as to obtain credit). We must, of course, trust the agent to perform issuance *only* under appropriate circumstances (for example, in conjunction with the segregation of certain "real" assets into a relevant bank account).

We observe that issuances and destructions remain indistinguishable from general transfers, under this scheme. Our approach differs from certain issuance mechanisms in that the total *capacity* for issuance is finite (limited by the capacity of the contract). Thus, the money supply cannot be *arbitrarily* increased.

3 Provably safe atomic swaps

An enterprise instantiation might deploy many Zether Smart Contracts, each tracking some particular virtual asset. Pairs of parties may wish to swap certain quantities of certain among these assets, in the absence of settlement risk.

We assume that some pair of parties has established the terms of a *desired* exchange—that is, "which assets" and "which amounts"—through a private channel (for example, Whisper could be used). We describe a system whereby these parties may orchestrate this exchange, under the guarantee that each payment will go through *only if the other does*, and only under the agreed-upon terms. We assume that the two would-be transactors have also agreed upon some (publicly sharable) session nonce, say nonce.

We propose a settlement contract, which alone shall be permitted to initiate Zether transfers. (This restriction does not materially hinder issuance and destruction, or other unilateral payments, as these can easily be "disguised" as bilateral transfers.) There are two, closely related, attacks against which a secure such settlement contract must guard. In one, after one party—Alice, let's say—submits her statement and proof to the settlement contract, the second party, let's say Bob, responds with a statement and proof which transfers Alice less than the amount upon which they had agreed. In the second, Bob simply *neglects* to respond to Alice's initial proof, before initiating a second "bilateral" exchange, in which he plays both sides; specifically, Bob reuses Alice's original statement and proof, together with a bogus statement and proof of his own which accords to Alice less than the amount specified by their agreement.

We describe a settlement contract which mitigates these attacks. For notational ease, we name the two assets at play currency and commodity. We require a Solidity datatype PendingSwap, which in turn invokes datatypes ZetherStatement and ZetherProof (we do not further elaborate on these latter datatypes, referring instead to the protocol). PendingSwap is constructed as follows:

```
1 struct PendingSwap
2 |   tuple ⟨ZetherStatement, ZetherProof⟩ currency
3 |   tuple ⟨ZetherStatement, ZetherProof⟩ commodity
```

We further define two global maps:

```
4 mapping(bytes32 ⇒ address) proofHashes
5 mapping(bytes32 ⇒ PendingSwap) pendings
```

We finally describe a *two-round-trip* protocol by which Alice and Bob can securely atomically swap.

1. Each party must, independently and asynchronously,
   (a) Generate a throwaway Ethereum address, and
   (b) Use this throwaway to send nonce, the asset type (say type), the party's own statement, and finally a *hash* of her own proof (say proofHash) to the contract. (If this transaction does not succeed—for example if she is front-run—then abort the procedure.)
2. Upon receiving any particular such (nonce, type, statement, proofHash) tuple, the contract must
   (a) Set proofHashes[proofHash] = msg.sender
   (b) Set pendings[nonce].type.statement = statement

   The contract must *also* ensure that these assignments cannot be overwritten or modified by any future transaction (as if they were final).
3. Each party must, independently, upon seeing *both* statement fields immutably populated,
   (a) Check whether the statement corresponding to the *opposite* asset type transfers to her the agreed-upon amount (she can check this by decrypting her own ciphertext (C_i, D)). If not, then abort.

(b) Send nonce as well as her proof to the contract, under the same throwaway used in step 1. The contract in turn requires that $\mathsf{proofHashes}[\mathsf{hash}(\mathsf{proof})] \stackrel{?}{=} \mathsf{msg.sender}$, and then sets $\mathsf{pendings}[\mathsf{nonce}].\mathsf{type.proof} = \mathsf{proof}$.

4. Once the contract has fully populated both $(\mathsf{statement}, \mathsf{proof})$ pairs (and all checks have passed), the contract sends both pairs to the relevant Zether contracts for execution. Optionally, all used mapping keys are deleted.

The idea of this protocol is that each party initially "locks" (a hash of) her proof against a throwaway (controlling its future use), and, before divulging its full contents, checks that she indeed stands to receive the promised amount from the counterparty. This check prevents the first attack described above; the lock prevents the second.

References

[BAZB] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. Unpublished manuscript.

[Dia19] Benjamin E. Diamond. Anonymous Zether. Unpublished manuscript, November 2019.