Anonymous Zether: Infrastructure

Benjamin E. Diamond
J.P. Morgan / Quorum

Abstract

We describe infrastructural utilities which may supplement enterprise deployments of Anonymous Zether.

1 Provably revealing oneself as the sender of a transaction

In general, in Anonymous Zether, *even the recipient* of a transfer necessarily possesses no means by which to identify the transfer's sender. (This is a consequence of the system's privacy characteristics.) In certain cases, this may be desirable; in other cases, the sender may wish to selectively reveal herself *to the recipient alone*, 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.

After the transaction is mined, the sender issues a *private* message to the recipient, which includes both the transfer's transaction hash and 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 receipt of this message, the recipient retrieves and parses the relevant transaction payload, uses r to decrypt the ciphertexts $\{(C_i, D)\}_{i=1}^N$, and finally confirms that the sender's ciphertext indeed transferred the received amount.

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

We note that the sender must trust the receiver: a leak of the viewing key r would irrevocably unmask the identity of the sender (and of the receiver) 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 shall 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 `acc` (account) ciphertext with an encrypted balance containing the full capacity of the contract (for example, $2^{32} - 1$). Her account shall be considered the "vault" account.

In order to privately issue tokens to some particular payee, this agent must initiate a transfer from the vault account to this payee. In order to privately destroy assets, a user simply transfers them back into the vault (and, perhaps, notifies 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 the relevant assets into some 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 enact exchanges among pairs of 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 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 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 reuses Alice's original statement and proof together with a bogus statement and proof which accords Alice less than their agreement specified.

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

```
struct PendingSwap
    tuple $\langle$ZetherStatement, ZetherProof$\rangle$ cash
    tuple $\langle$ZetherStatement, ZetherProof$\rangle$ collateral
```

We further define two global maps:

```
mapping(bytes32 $\Rightarrow$ address) proofHashes
mapping(bytes32 $\Rightarrow$ PendingSwap) pendings
```

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

1. Each party must, independently,
   (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 each such (nonce, type, statement, proofHash) tuple, the contract must
   (a) Set proofHashes[proofHash] = msg.sender
   (b) Set pendings[nonce].type.statement = statement

   The contract *also* ensures that these assignments can not be overwritten or modified by any future transactions (as if they were final).
3. Each party must, independently, upon seeing *both* statement fields immutably populated,
   (a) Check whether among the two statements resides one which 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 the nonce as well as her proof to the contract, under the same throwaway used in step 1. The contract in turn requires that $\mathsf{proofHashes}[\mathsf{hash}(\mathsf{proof})] == \mathsf{msg.sender}$, and then sets $\mathsf{pendings}[\mathsf{nonce}].\mathsf{type}.\mathsf{proof} = \mathsf{proof}$.

4. Once the contract has fully populated both such $(\mathsf{statement}, \mathsf{proof})$ pairs (and all checks have passed), the contract sends both 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.