Quorum - ZSL Integration: Proof of Concept

Technical Design Document

This version of the Technical Design Document has been edited for public release, and is provided for background information only. Pseudocode and code examples presented in this document are likely to have been implemented differently. Please refer to the deployed code at https://github.com/jpmorganchase/zsl-q

1. Introduction & Background

J.P. Morgan (JPM) has developed Quorum, an Ethereum-based distributed ledger protocol, designed to provide the Financial Services Industry with a permissioned implementation of Ethereum that supports transaction and contract privacy, through the use of Public Contracts (which are executed in the standard Ethereum way, and are visible to all participants in the distributed ledger) and Private Contracts (which are shared between the parties to the private contract using a pluggable component called Constellation, but can not be read by other participants).

The current Quorum design / implementation is subject to a number of limitations, including:

- The Private Contract system does not currently support mass conservation / prevention of double spend
- Quorum does not allow a private contract to write to a public contract. Only reads of public contracts are allowed in private contracts. Similarly, a public contract cannot access the state of private contracts

ZSL is a distributed ledger agnostic protocol designed by the Zerocoin Electric Coin Company (ZcashCo), which uses zk-SNARKS to enable the transfer of digital assets on a distributed ledger, without revealing any information about the Sender, Recipient, or the quantity of assets that are being transferred, while ensuring that:

- the Sender is authorized to transfer “ownership” of the assets in question,
- the assets have not been spent previously (i.e. prevention of double-spend), and
- the transaction’s inputs equal its outputs (mass conservation).

2. Objectives & Scope

The objective of this document is to describe how a proof of concept (POC) implementation of ZSL on Quorum will be delivered.

The POC implementation is intended to help JPM and other interested parties assess:

- the feasibility of adding ZSL to Quorum,
- the advantages such an integration would confer (with a particular focus on addressing the limitations listed above), and
- any constraints ZSL would impose.

Requirements

This document is intended to provide guidance regarding:

- which requirements the POC will meet, and
- which requirements the POC will not meet but could be met at a future stage of development.

The POC solution described in this document will:

a. support the private, confidential transfer of assets to satisfy obligations arising from private contracts,

b. allow private contracts to verify that the transfer of assets has taken place,

c. prevent the disclosure of (i) the Sender's and Receiver's identities, and (ii) the quantity of assets being transferred, to other participants in the distributed ledger,

d. prevent the Recipient of an asset transfer from ascertaining who the Sender received the assets from, by examining the distributed ledger,

e. ensures mass conservation (i.e. the supply of a particular asset can only be increased by a party who is authorised to do so),

f. prevent double-spending,

g. be compatible with Quorum's existing regulatory oversight mechanism (i.e. including the Regulator as a party to the private contract will allow the Regulator oversight of the ZSL transactions that relate to that contract);

h. support the example equity trade use case described in Appendix A, and

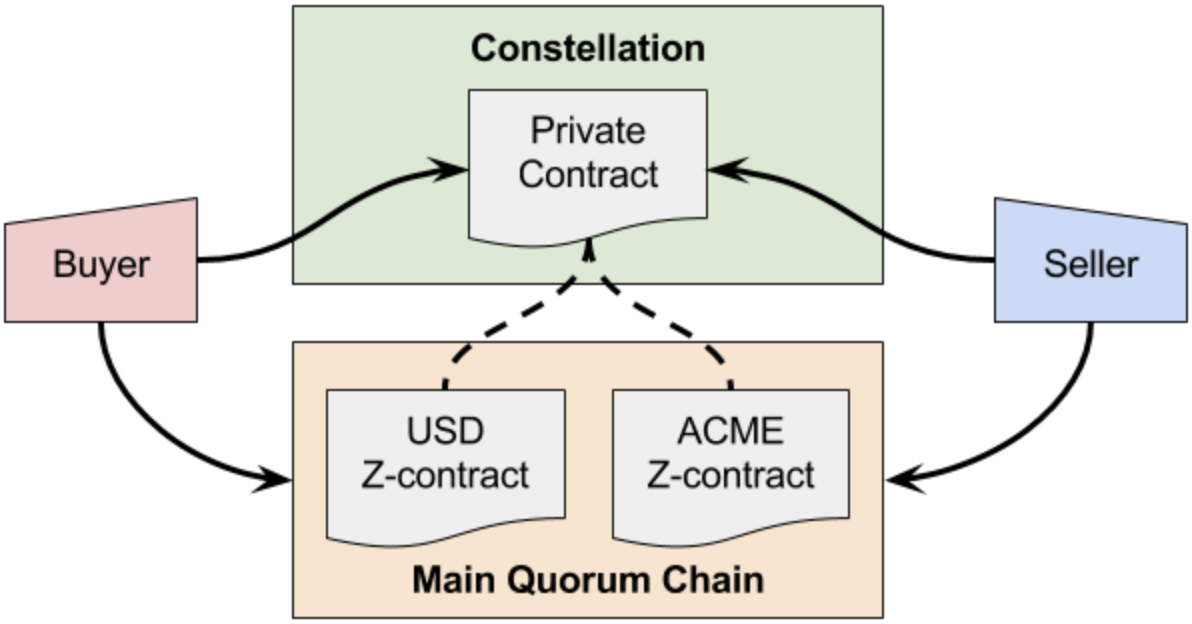
i. minimise drift from the public Ethereum codebase.

The POC solution will not support cryptographically-assured DvP (i.e. atomic exchange of assets). However, it will be possible to add DvP functionality at a later stage (see section 5.1).

The POC solution is not intended to demonstrate high performance or throughput. See section 4.2 for more information on performance and scalability limitations.

3. Solution Overview

The approach we propose for the POC retains the use of private contracts (deployed using Constellation) for describing trades (i.e. the business logic), and adds ZSL-enabled smart contracts (z-contracts) for settlement.



Tokenized assets (representing cash, securities, or other tradable instruments) will be issued using z-contracts, which will support shielded transactions (i.e. the sender, recipient and quantity of assets being transferred remain confidential). There will be a 1:1 relationship between z-contracts and tokenized assets (e.g. one z-contract for tokenized USD, another z-contract for tokenized ACME shares). Z-contracts will persist on the main Quorum ledger, and will support settlement of multiple trades independent between multiple pairs of users[1].

Each trade will be represented by a private contract, which define the type of trade, the counterparties, the underlying assets, the price(s) that has been agreed, and any other relevant parameters or business logic that are relevant to the trade.

For the purposes of the POC, we will focus on demonstrating the lifecycle of a vanilla securities trade, where z-tokens representing US Dollars are exchanged for z-tokens representing a security. In the example below (which is laid out in more detail in Appendix A), the parties are executing a trade of ACME shares.

[1] For the purposes of the POC, we assume that each trade will involve two parties. However, in principle, this approach can be extended to support more than two parties to a trade.

1. One of the Parties will create a Private Contract, defining the terms of the trade[2]. In this use case, the buyer proposes
2. The other Party accepts the terms of the trade.
3. The Private Contract instructs the Buyer to make payment to the Seller.
4. The Buyer makes the payment by executing a shielded payment of USD z-tokens.
5. The Buyer notifies the Private Contract that payment has been made.
6. The Private Contract verifies that payment has been made.
7. The Private Contract instructs the seller to deliver the securities.
8. The Seller makes delivery by executing a shielded payment of ACME z-tokens.
9. The Seller notifies the Private Contract that delivery has been made.
10. The Private Contract verifies that delivery has been made, at which point the trade is considered to have been settled.

The business logic flow outlined above is merely an example. The business logic is contained entirely within the private contract, meaning that a wide variety of trades can be executed and settled by creating private contracts with the relevant business logic.

In principle, data sourced from outside the ledger (e.g. LIBOR data for interest rate swaps, default events for credit default swaps) can be incorporated into the business logic through the use of digitally signed messages or oracles. However, this functionality is outside the scope of the POC.

Note that ZSL does not currently support DvP-style settlement. However, we are confident that it will be possible to add support for DvP in the future.

The approach outlined above supports Quorum's existing regulatory oversight model. If a regulator is a party to the private contract, they can independently verify the shielded payments and transfers of z-tokens.

It is important to note that the solution being proposed is a proof of concept, intended to demonstrate the technical and cryptographic principles involved, and explore ZSL's suitability for JPM's expected use cases. Some components will not be as tightly integrated as they would be in a production environment, necessitating some manual configuration and actions. Further development, integration and testing will be required to enhance the solution such that it is suitable for deployment in a production environment.

[2] For the purposes of the POC, we assume that the terms of the trade have been previously agreed (e.g. verbally or using some kind of e-trading facility).

Technical Components

For the purposes of this document, the entire transaction that is intended to occur between the parties involved is described as a *trade.*

Appendix A describes a simple equity trade use case, which we will refer to throughout this document. Note that the lifecycle of this equity trade has been deliberately simplified - its goal is to demonstrate a simple lifecycle, rather than accurately model the lifecycle of a real trade.

The POC will use a simplified adaptation of the Zcash protocol. The most notable difference is that the note details will be provided to the Recipient via the Private Contract, instead of being encrypted and attached to the transaction itself.

Appendix B outlines the cryptographic elements which will be involved at each of the first six stages of the equity trade use case.

In this section, we describe in greater detail the key elements that must be implemented for the POC: the z-contract, private contract, zk-SNARK prover, and zk-SNARK verifier.

3.1. Z-contract

We use the terms *z-contract* and *z-token* to refer to ZSL-capable contracts, and the tokens issued using such contracts.

A z-contract is a contract on the main Quorum chain, which supports the issuance of z-tokens that can be exchanged both transparently, and privately using ZSL technology. It can be used by multiple users.

Each z-contract will have his own type of z-token representing some financial instrument. Z-tokens can represent currency (e.g. USD), securities (e.g. ACME shares, bonds), or other financial instruments which can be represented as single units (e.g. futures, options).

The supply of z-tokens for a given z-contract can be defined and controlled in a number of ways. For the POC, the supply will be fixed when the z-contract is created. For a production implementation, a party could be granted authority to increase the supply by issuing new z-tokens.

When initially issued, z-tokens will be “transparent” - i.e. they will be visible on the ledger, just like standard ERC20 tokens. In order to transact z-tokens in a private and confidential manner, they must be “shielded”, which involves creating a “note commitment”, which can then be used by the note’s recipient as the input to a shielded transaction. Each note represents a specific number of tokens, which can be sent to another party (or unshielded) by the note’s recipient.

For the POC, the z-contract will support a subset of ERC20 functions for controlling transparent tokens. New functions will be added that relate to the following ZSL-specific operations:

- “shield” z-tokens into an accumulator (i.e. create a new note, reducing their transparent token balance and increasing their shielded token balance by the same amount),
- “unshield” notes from the accumulator back into the transparent account (i.e. spend a note, decreasing their shielded z-token balance and increasing their transparent z-token balance), and
- create a “shielded transaction”, which transfers the shielded tokens to another participant (i.e. consumes a note, decreasing the Sender’s shielded z-token balance, and increasing the Recipient’s shielded z-token balance).

These ZSL operations provide strong privacy: shielding does not reveal the recipient, unshielding does not reveal the input note, and shielded transactions reveal neither the recipient nor the input note(s). Shielded transfers also conceal the amount being transferred (i.e. the number of z-tokens).

In order to achieve these privacy and confidentiality features, the z-contract maintains some state:

1. The note commitment accumulator is a merkle tree data structure which records commitments to all of the notes that have been created so far. This tree must be of a fixed depth. For the POC, it will have a depth of 29, which will support a capacity of over 250m transactions. (See section 4.2 for further information on capacity limitations.)
2. A list of “nullifiers” that is used to prevent double-spending
3. A historic list of accumulator states (merkle tree roots) that proofs are authorized to spend from.

Each ZSL operation requires a zk-SNARK proof to be verified for all of its operations. Each ZSL operation is slightly different, and requires a different statement to be proven by the zk-SNARK. This means that three different arithmetic circuits (which encode the underlying shielding logic) will be written, and three different zk-SNARK verifiers will be provided for the POC.

zk-SNARKs, by their nature, require a set of parameters for use in generating and verifying proofs. These parameters are specific to the statement being proven. For the POC, these parameters will be generated by ZcashCo. A production deployment would require that these parameters be generated securely, likely using a secure multi-party computation protocol similar to that used to generate the Zcash parameters.

The z-contract will implement a constant function that verifies whether a specific note exists. This will be used to allow the private contract to verify that a shielded transaction has taken place (see s4.2.2).

See Appendix B for an overview of how shielded transactions will be created and verified within the context of the equity trade use case.

3.1.1. Note Tracker

For the purposes of the POC, a simple note tracker will be implemented in the Quorum client. It will perform the following functions:

- manage the spending key, from which the shielded payment address is derived,
- keep track of notes that have been received across each z-contract, and
- track the total balance of the user's unspent notes across each z-contract.

The note tracker will be subject to the following restrictions:

- While it will automatically detect and record notes that have been provided via a private contract, notes transmitted by other means must be added manually.
- It will not support the use of more than one input note in a shielded transfer. As a result of the latter, it will be necessary to manually merge notes.

If the project advances beyond the POC phase, the simple note tracking functionality described here will need to replaced with a more functional "wallet"-style solution.

3.2. Private Contract

The parameters of the trade (e.g. the buyer, the seller, the type of trade, the underlying instrument, settlement instructions) and the trade's lifecycle (i.e. the various states that the trade transitions through until the trade is finally settled) will be defined in the private contract.

From a technical perspective, the underlying instruments (ACME shares and USD in our equity trade use case) will be identified as z-tokens issued by a specific z-contract (i.e. the private contract will identify the relevant z-contract by its address).

The private contract will implement a finite state machine to track progress of the trade's lifecycle. Moving from one trade state to the next will only be possible when a specific trade event occurs. Execution of a private contract function to perform an action and trigger a trade event will be restricted to the correct state.

For example, if Alice creates the private contract described in Appendix A, and initializes it into the Bid state, the only function a potential counterparty can call is `acceptBid()`[3] which changes the trade state from "Bid" to "Done". Any other function will return an error.

Example:

```
contract EquityTrade {
    enum TradeState { Bid, Done, PaymentReceived, Settled }
    TradeState state;
...
```

[3] As previously noted, the equity trade lifecycle has been deliberately simplified. A production implementation would likely allow a potential counterparty to reject the bid.

```
    function acceptBid() returns (bool result) {
        if (state != TradeState.Bid) throw; // the action to
accept a bid is only permitted when in the Bid state
        state = TradeState.Done;
        LogDoneEvent(msg.sender, arg1, arg2,...);
        return true;
    }
```

3.2.1. Instructing a Shielded Transaction

A key limitation of Quorum is that a private contract is unable to write to a public contract. The POC solution will retain this limitation. However, it will include a mechanism to allow the private contract to "instruct" a party to carry out a specific shielded transaction. This instruction will be received and interpreted by the party's Quorum client as described below.

When the trade lifecycle reaches a state at which one party is expected to make a payment or transfer some assets to the other party, the private contract will "instruct" the relevant party to create a shielded transaction by using Ethereum's contract events and logging functionality. The party who is to create the shielded transaction will listen for an event emitted by the private contract; upon receipt of that event, they are expected to create the shielded transaction.

For the purposes of the POC, this will involve manually adding a JavaScript event listener and callback to each party's Quorum client, which is executed when the private contract emits a relevant event. For example, in our equity trade use-case example, Alice will add an event listener (e.g. by loading and running a .js file) that is executed when the private contract emits a Done event. The callback will use the arguments logged with the Done event to begin the process of creating a shielded transaction.

In JavaScript pseudo-code:

```
var event = private_contract.LogDoneEvent();
event.watch(function(error, result) {
    if (!error) {
        // select input note, generate proof, create transaction
    }
});
```

In more detail, this involves the following steps:

i. The private contract emits a Done event, with the following information:
   - *rho* (a pseudo-random number for the note commitment),
   - the address of the relevant z-contract (e.g. the USD z-contract)
   - the quantity of z-tokens to be transferred (e.g. the amount of USD to be paid), and

- the recipient's shielded payment address.

ii. The event listener configured in the Quorum client detects the event and triggers the creation of a shielded transaction.

iii. The Quorum client obtains an input note of suitable value from the integrated note tracker.

iv. The Quorum client obtains a witness from the z-contract. (In order to create the proof, the user must obtain a witness into the accumulator to demonstrate knowledge/existence of a note that they are authorized to spend. The z-contract will provide a constant function that accepts a commitment in the accumulator and returns a witness.)

v. The Quorum client invokes the zk-SNARK proof generator using the JavaScript ZSL Demo API with the arguments provided by the private contract, the input note obtained from the note tracker, and the witness obtained from the z-contract.

vi. The Quorum client submits the proof to the z-contract. In pseudo-code, this might look something like:

```
z_contract.addShieldedTransaction(proof, ...)
```

The z-contract will verify the proof, and update the note commitment accumulator (at which point, the transaction can be verified).

In order to verify the proof, the z-contract will invoke one of the three built-in precompiled contracts (e.g. `zsl_verifyShieldedTransaction(...)`, which perform zk-SNARK verification using the C++ library libsnark.

It is important to note that the private contract cannot enforce settlement of a trade. For example, in our equity trade use case, Bob could choose not to deliver the ACME z-tokens to Alice after she has paid him the USD z-tokens. For this use case (and similar use cases that involve the atomic exchange of assets), the risk of failure to deliver can be mitigated by adding support for DvP to ZSL (see section 5.1).

3.2.2. Verifying that a Shielded Transaction took place

In the equity trade use case, the private contract will verify that a shielded transaction has taken place at steps 6 and 10.

Given a note (v, rho, a_{pk}), the private contract will call a z-contract constant function, e.g. `verifyNote(...)`, to check that the note commitment derived from the note is found in the commitment accumulator.

3.3. zk-SNARK Prover

The creation of z-transactions requires the generation of a zk-SNARK proof by the Sender, who then includes the proof as part of the transaction that she broadcasts to the network.

For the POC, the zk-SNARK prover will be a C++ executable. The Quorum client will invoke the prover executable which takes command-line parameters and returns the proof.

To reduce complexity, the POC solution will use three separate zk-SNARK provers, one for each of the ZSL operations (i.e. shielding, unshielding and creating a shielded transaction).

Invoking the zk-SNARK Prover from the Quorum cllent

The Quorum client will invoke the prover by using the ZSL Demo API, a new JavaScript API, which enables JavaScript callbacks triggered by Quorum events to call functions implemented in Go.

The implementation of the JavaScript ZSL Demo API under the 'zsl' namespace will follow the design pattern used to implement the Quorum API under the 'quorum' namespace.

This will ensure that:

1. both Quorum and ZSL can continue to update and expand their APIs in their own respective namespace, so there is no conflict with other JavaScript APIs inherited from upstream Geth, and
2. the API can be implemented in native Go, offering access to a rich standard library, improved native performance over interpreted JavaScript, and the flexibility of using Go's foreign function interface to enable use of components written in other languages (such as proof verification using libsnark which is written in C++).

In practice, to create the ZSL Demo API, modifications will be made to:

- `core/quorum/api.go`
- `eth/backend.go`
- `internal/web3ext/web3ext.go`
- `rpc/server.go`

When launching the Quorum client, the 'zsl' module can be specified via the `--rpcapi` command line option.

For the POC, we assume that a single instance of a Quorum client will support only a single user who will use the same spending key, and thus the same shielded address, across all z-contracts the user participates in.

The ZSL Demo API will include commands to create shielded transactions. For example:

- `zsl.getNewAddress()`
- `zsl.createShielding(...)`
- `zsl.createUnshielding(...)`

3.4. zk-SNARK Verifier

Confirming the validity of z-transactions requires that all ledger participants (i.e. not just the parties to the z-transaction) be able to verify zk-SNARK proofs.

The obvious way to add this functionality is to create a new opcode. However, this would necessitate changes to both Solidity and the EVM, that would conflict with the requirement to minimize drift from the core Ethereum codebase.

As an alternative to adding a new opcode, we propose adding native code to the Quorum client that can verify zk-SNARKs (i.e. a built-in), assigning it a fixed contract address in the range `0x000....000400` to `0x000....0fffff`[4], and treating it as a precompiled contract. This will allow the z-contract to access a built-in precompile using the abstract function interface.

To reduce complexity, the POC solution will use three built-in precompiles - one each for shielding, unshielding and shielded transactions.

Note that a change included in Solidity 0.4.0[5] throws an error if no code is found at a contract's address. It is currently possible to work around this restriction by including the precompiled contract's address in genesis.json.

Future changes to Ethereum may render this workaround ineffectual. An alternative workaround is to add zk-SNARK verification functionality to an existing built-in precompiled contract (e.g. sha256) which gets called if the data is prefaced with a specific string (e.g. "zsl://").

The zk-SNARK verification functionality will be provided by the C++ library `libsnark`. The POC will integrate this library into the Quorum client by using Go's foreign function interface.

3.5. Target Platform

The target platform will be 64-bit Linux. The project will be developed on Ubuntu Linux but ZcashCo expect that it will be possible to compile and run the source code on RHEL.

ZcashCo will base the POC on the current (as of May 2017) version of Quorum (which is based on version 1.5 of the go-ethereum implementation of the Ethereum protocol). Where appropriate, ZcashCo will implement the POC in such a way as to minimize the effort required to upgrade to version 1.6.6 of go-ethereum.

[4] See https://github.com/ethereum/wiki/wiki/Consortium-Chain-Development

[5] See https://github.com/ethereum/solidity/blame/develop/Changelog.md#L202

4. Caveats & Disclosures

4.1. Limited Anonymity Set

The privacy and confidentiality provided by ZSL relies on there being a large enough anonymity set to ensure that a third party observer cannot deduce the Sender, Recipient or quantity of z-tokens being transferred through techniques such as a process of elimination, correlation of on-ledger transactions with off-ledger events, monitoring which IP addresses transactions are broadcast from, etc.

For a POC, a limited number of parties will be using the distributed ledger, so the anonymity set will be comparatively small. It is assumed that, in a real deployment scenario, there will be sufficient transactions to ensure a large enough anonymity set. If this turns out not to be the case, other solutions can be explored (e.g. creating empty transactions to increase the site of the anonymity set).

4.2. Performance & Scalability Limitations

The confidentiality and privacy provided by ZSL comes at a cost to performance. Generating the zk-SNARK proofs that are required to create a shielded transaction is a computationally- and memory-intensive operation. Each shielded transaction requires the creation of at least two note commitments (normally one primary output and one change output), which are stored in a merkle tree data structure (also referred to above as the z-contract's note commitment accumulator). The action of generating the two zk-SNARK proofs required by the pair of note commitments is dubbed a "joinsplit".

The depth of the merkle tree defines the maximum number of note commitments that can be stored before action must be taken to either extend the merkle tree (by increasing its depth) or migrate to a "new" z-contract (with a new and empty merkle tree).

The maximum number of joinsplits that can be supported by a merkle tree of a given depth d is calculated as $2^{(d-1)}$. For example, a depth of 21 would support a maximum of 1,048,576 joinsplits, while a depth of 29 (which is what Zcash uses, and what we plan to use for the POC) will support 268,435,456. However, the greater the depth, the greater the complexity of the joinsplit operation. There is a roughly linear relationship between the merkle tree's depth and the time it takes to complete a joinsplit operation.

At present (6th March), using our standard benchmarking for Zcash (which uses a depth of 29), a joinsplit operation takes 42.6 seconds on an Intel Xeon E3-1225 v2 3.2GHz processor (4 cores) and requires nearly 3GB of RAM. (Note that this is the time required to create the zk-SNARK proof; verifying the proof takes just 28 milliseconds and requires just over 6MB of RAM.)

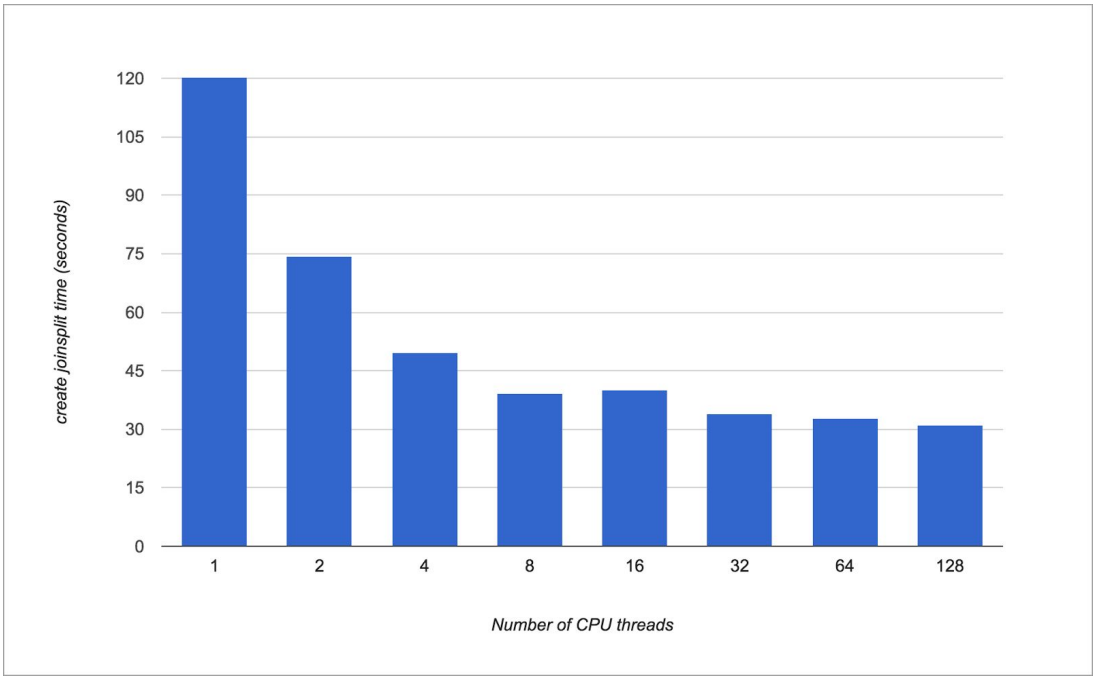
Because of the changes that are required to implement ZSL on Quorum will require some changes to the cryptographic circuit. The impact of these changes (in terms of the

computational time required to generate the zk-SNARK proof necessary to send a shielded transaction) will not be clear until they have been implemented and tested. However, we are confident that it will be possible to achieve performance that is similar to Zcash's.

It is important to note that the time required to carry out the joinsplit operation will not impact the performance or throughput of the Quorum ledger. Its impact is limited to the amount of time it takes the generate the zk-SNARK proof(s) necessary to carry out a shielded transction.

Hardware Performance

In December, we benchmarked the joinsplit operation on a 128-core Amazon EC2 instance, which (according to the information reported by the operating system) was running on multiple Intel Xeon E7-8880 v3 2.30GHz processors (18 cores, with hyperthreading, which allows two threads to run on each core). We recorded the time taken to generate a joinsplit, with the software limited to using various numbers of threads. The results are graphed below, and illustrate how the performance improvement flattened out as more threads were enabled.



We are unable to explain why the result for 16 threads is anomalous. One theory is that, given the hardware is shared, another user's process may have been running on the same processor as the benchmarking software. The only way to eliminate this as a factor would be to run the benchmarks on hardware that we could control for the duration of the test.

Note that the processor's clock speed has an impact on performance. At the time of this test, our standard benchmark, which runs on a four-core 3.2GHz CPU, was 44.6 seconds, while using the same number of cores on a 2.30GHz processor took 49.8 seconds.

Cryptographic Optimisation

We are currently conducting research into the possibility of optimising the cryptographic implementation of zk-SNARKs creation (i.e. the underlying mathematical implementation, which takes the form of a complex arithmetic circuit). Early indications suggest that this could yield significant performance improvements. We hope to be able to confirm this and publish the results later this year.

Such optimizations will be applicable to ZSL on Quorum (but would require a reimplementation of the zk-SNARK circuit, proof generator and verifiers).

4.3. Randomness

It is necessary for a private contract to generate *rho*, a pseudo-random number used to generate the note commitment. This acts as a "challenge" to prevent replay attacks against the private contract where a note for a different private contract is submitted.

For the POC, to generate *rho* the following will take place:

1. When Alice creates a trade, she will instantiate a private contract and pass in some randomness, r_alice. For the POC, this randomness can be generated by using the Quorum console and calling Math.random (the client's JavaScript runtime has been recently updated to use a seeded RNG).
2. When Bob accepts the terms of the trade, he will call the private contract's acceptBid() function and pass in some randomness, *r_bob*.
3. The private contract's acceptBid() function will compute *rho = H(r_alice | r_bob)* where *H* is a hash function, such as the built-in sha256.

It is important for the private contract's computation of *rho* to be deterministic so that the private contract's state is always reproducible given the inputs.

5. Post-POC Functionality

To minimize complexity, lead time and delivery risk, the scope of the the POC solution's functionality is limited. However, if the POC is successful, two specific areas of functionality should be considered.

5.1. Support for DvP

As specified above, the POC solution will not support cryptographically-assured DvP (i.e. atomic exchange of assets). This is because ZSL currently has no means of supporting shielded DvP-style functionality.

We are currently researching whether it is possible to add this functionality, by upgrading the zk-SNARK cryptographic circuit to support shielded hash time-locked contracts (HTLC). If this research is successful, it would allow parties to place z-tokens into a type of shielded escrow, which unlocks and delivers the z-tokens to the recipients simultaneously.

In the context of the equity trade use case referred to throughout this document, steps 3 thru 10 would be replaced with the following steps:

3. The Private Contract instructs Alice to place the USD z-tokens into escrow.
4. Alice commits the relevant amount of USD z-tokens to a shielded hash timelocked contract.
5. Alice provides the Private Contract with evidence that the necessary funds have been placed in escrow, and specifies both the hash used and the timeout period.
6. The Private Contract verifies that the funds have have been placed in escrow.
7. The Private Contract instructs Bob to place the USD z-tokens into escrow.
8. Bob commits the relevant amount of ACME z-tokens to a shielded hash timelocked contract using the hash provided by Alice but with a shorter timeout period.
9. Bob provides the Private Contract with evidence that the assets have been placed in escrow.
10. The Private Contract verifies that the assets have have been placed in escrow.
11. The Private Contract instructs Alice to claim the assets by disclosing the hash preimage she used.
12. Alice claims the ACME z-tokens by disclosing the hash pre-image.
13. Bob uses the hash pre-image disclosed by Alice to claim the USD z-tokens before the HTLC transaction times out.

This approach would preclude Bob from refusing (or being unable) to deliver the ACME z-tokens to Alice after having received the USD z-tokens from her. With HTLC, if Bob failed to place the ACME z-tokens, Alice would simply recover her USD z-tokens by letting the timelock time out.

5.2. Support for other trade types

As mentioned in section 3, with the solution described herein, the business logic for the trade resides entirely within the private contract. The ZSL elements are used purely for settling obligations that arise from the private contracts.

As a result, different types of private contract can be developed to enable different trade types, without requiring any changes to ZSL.

It should be noted that, for most types of derivatives, it will be necessary to develop private contracts that can receive data from oracles, or can react to input presented in the form of a digitally-signed message to enable certain types of trades. Examples include:

- date-and-timestamps for trades in which the passage of time forms part of the business logic (e.g. options, forwards, swaps)
- market data for derivatives trades who business logic depends on the price of an underlying security (e.g. cash-settled futures) or a benchmark (e.g. interest-rate swaps, floating-rate notes)
- Event-type triggers (e.g. notification of a default in relation to a credit default swap's reference obligation).

ZSL can support the settlement of obligations arising from most, if not all, such derivatives (assuming the relevant asset is available as a z-token).

5.3. Extension of visibility

Constellation does not currently support adding a new party to a private contract after it has been initialized and transactions have occurred, and allowing the new party to view past transactions involving that private contract.

The POC solution will not add such functionality. However, in principle, ZSL is likely to be compatible with such functionality if it is added in the future, as the ability to verify that shielded transactions have taken place is available to any party that has access to the note details (i.e. v, rho, and a_{pk}).

In other words, if a method of granting a new party access to view the past transaction history of a private contract, can be implemented, we see no reason why the new party would not be able to see see and verify the details of past shielded transactions executed as part of that private contract.

6. Intellectual Property

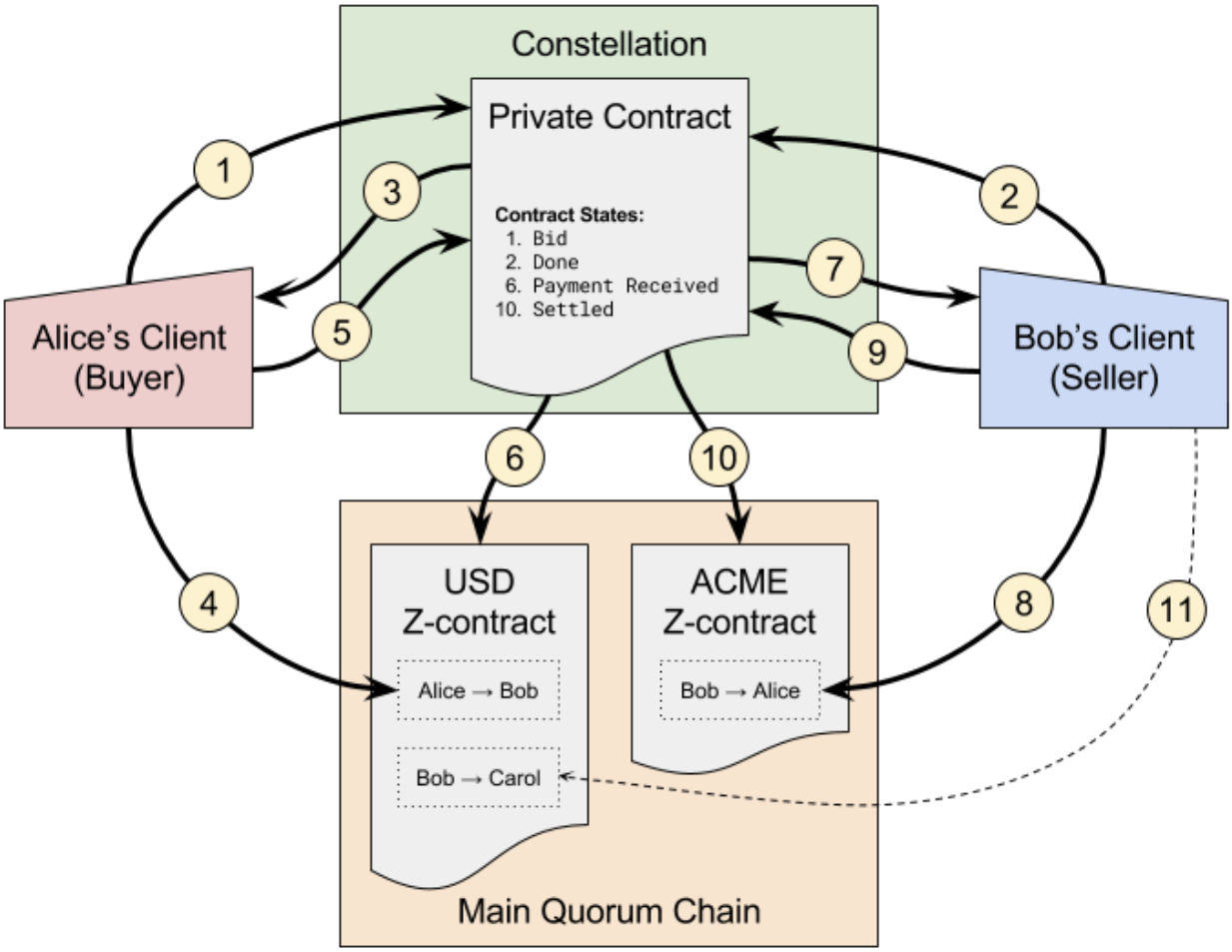
ZcashCo may create various software components for the POC by reusing or creating a derivative work of the Zcash source code, which is currently licensed as described in https://github.com/zcash/zcash/blob/master/COPYING. Where possible (i.e. where the code being reused or adapted is copyright of ZcashCo), ZcashCo will license such software components under the Apache License (version 2.0).

ZcashCo will retain copyright of the source code for any components it creates for the POC and will license those components under the Apache License (version 2.0).

ZcashCo may use, adapt or create a derivative work of other open source software for use in the POC, in which case such software will retain its original license.

For the avoidance of doubt, the Apache License (version 2.0) grants a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, sublicense, and distribute source code licensed under said license.

Appendix A: Equity Trade Use Case (v4)



Beginning State:

- Z-contracts have been created for US dollars (the USD z-contract) and ACME shares (the ACME z-contract),
- Z-tokens have been issued into both contracts by the relevant issuer, then shielded and transferred to Alice and Bob.
- Alice owns some USD z-tokens, and Bob owns some ACME z-tokens. Both their holdings are shielded (i.e. a third-party observer cannot tell who owns what).

User Story

1. A Private Contract is established between Alice and Bob using Constellation.
    a. The Private Contract specifies an equity trade of a specific quantity of ACME shares at a specific price in USD, between two specific parties: Alice (who is buying the ACME shares) and Bob (who is selling ACME shares).

   b. The Private Contract references the USD and ACME z-contracts, and the relevant public keys and payment addresses of the parties.
   c. One party initialises the contract (this is the equivalent of bidding/offering). It doesn't matter which party does this - in this example, it's Alice.
   d. After being initialised, the contract state is "`Bid`" (it would be "`Offer`" if Bob had initialised it).

2. The other party sends the Private Contract a transaction indicating acceptance of the terms.
   a. In this example, it is Bob who accepts Alice's bid.
   b. At this point, the trade is "done" (i.e. the terms are agreed and both parties have committed to the trade) and all that remains is for Settlement to take place. Assume that the USD must be paid first.
   c. Contract state: `Done`.

3. The Private Contract instructs Payment.
   a. When the contract's status updates to Done, it issues an instruction to the Buyer's (i.e. Alice's) client to pay the relevant amount of USD to the Seller (Bob).
   b. Alice's client receives and queues that instruction, and instructs a shielded payment.

4. The Buyers pays USD to the Seller.
   a. Alice pays the relevant amount of USD z-tokens to Bob's USD payment address by generating the necessary zk-SNARK proof and sending it to the USD z-contract.
   b. A shielded transaction takes place, creating a note within the z-contract which only Bob can spend (i.e. Bob's USD z-token balance is increased).
   c. Alice's balance of USD z-tokens is reduced accordingly.

5. The Buyer provides evidence of payment to the Private Contract.
   a. Alice sends the Private Contract a transaction with the output note of the USD payment.
   b. This also transmits the note to Bob so he can spend the note.

6. The Private Contract verifies the payment.
   a. The Private Contract calls a constant function on the USD z-contract, using the note supplied by Alice, to verify that the payment is valid.

   b. The z-contract responds in a binary fashion to indicate whether the note commitment is in the z-contract's note accumulator (in which case the shielded payment is valid) or not.
   c. If it is valid, the contract's status updates to `Payment Received`, and...

7. ..the Private Contract instructs Delivery.
   a. The Private Contract issues an instruction to the Seller's (i.e. Bob's) client to transfer the relevant amount of ACME shares to the Buyer
   b. Bob's client receives and queues that instruction, and prompts him to make the payment.

8. The Seller delivers ACME shares to the Buyer.
   a. Bob transfers the relevant amount of ACME z-tokens to Alice's ACME payment address by generating the necessary zk-SNARK proof and sending it to the ACME z-contract.
   b. A shielded transaction takes place, creating a note output that only Alice can spend (i.e. Alice's ACME z-token balance is increased).
   c. Bob's balance of ACME z-tokens is reduced accordingly.

9. The Seller provides evidence of delivery to the Private Contract
   a. Bob sends the Private Contract a transaction with the output note of the ACME delivery.
   b. This also transmits the note to Alice so she can "spend" the note (i.e. transfer those tokens to someone else).

10. The Private Contract verifies delivery.
   a. The Private Contract calls the ACME z-contract (using a constant function), using the note supplied by Bob, to verify that the transfer is valid.
   b. If it is valid, the contract's status updates to `Settled`.

11. After Alice has delivered the USD z-tokens to Bob in step 5, he can send them to a third party (e.g. Carol).
   a. Carol will not be able to ascertain the source of the tokens (i.e. that Bob obtained them from Alice).
   b. Alice will not be able to ascertain when Bob transfers the tokens to someone else (or who the recipient is). She will be able to see that a transaction has occurred (because the transaction is written to the z-contract on the main

Quorum chain which she has access to) but she will not be able to ascertain the Sender, Recipient, nor the quantity of tokens being transferred.[6]

The same holds true for the ACME z-tokens Alice has obtained from Bob.

[6] It should be noted that if Alice, Bob and Carol are the only parties using that z-contract (and Alice is aware of this), then Alice would be able to deduce that a transaction to which she is not a party must involve Bob and Carol.

Appendix B: Overview of Proposed ZSL / Quorum POC Cryptographic Implementation

Diagram v0.4

1

Alice (the Buyer) supplies the following when she initializes the private contract:

- Payment z-contract ID (i.e. the USD z-contract)
- Quantity of payment z-tokens (i.e. number of USD - v_1)
- Equity z-contract ID (i.e the ACME z-contract)
- Quantity of equity z-tokens (i.e. number of ACME shares - v_2)
- Alice's payment address (i.e. Alice's public key - a_{pk})

Implicitly, the price Alice is bidding is v_1 divided by v_2

2

Bob (the Seller) supplies his payment address (i.e. Bob's public key - b_{pk}) when he accepts Alice's bid.

3

Once Bob has accepted Alice's bid (and the private contract has been updated to state Done) the private contract instructs Alice to make a shielded payment, specifying the following parameters:

- Payment z-contract ID
- Quantity of payment z-tokens (v_1)
- Bob's payment address (b_{pk})
- A pseudo-random number (rho_{new})

4

In response to the instruction from the private contract, Alice:

a. uses the commitment scheme COMM(v_1,b_{pk},rho_{new}) to generate a commitment (cm_{new}) for the output note being sent to Bob,

b. uses the commitment scheme COMM(v_{change},a_{pk},rho_{change}) to generate a commitment (cm_{change}) for a note sending any change back to herself (Alice selects rho_{change} herself),

c. calculates a blocker for each new note using the SHA256 hash of 0x00, her secret key (a_{sk}) and rho for both the output note being sent to Bob (rho_{new}) and any change output note (rho_{change}),

d. calculates the input notes' nullifiers (nf_i^{old}) using the SHA256 hash of 0x01, her secret key (a_{sk}) and the rho_{old} for each input note, and

e. generates the zk-SNARK proof.

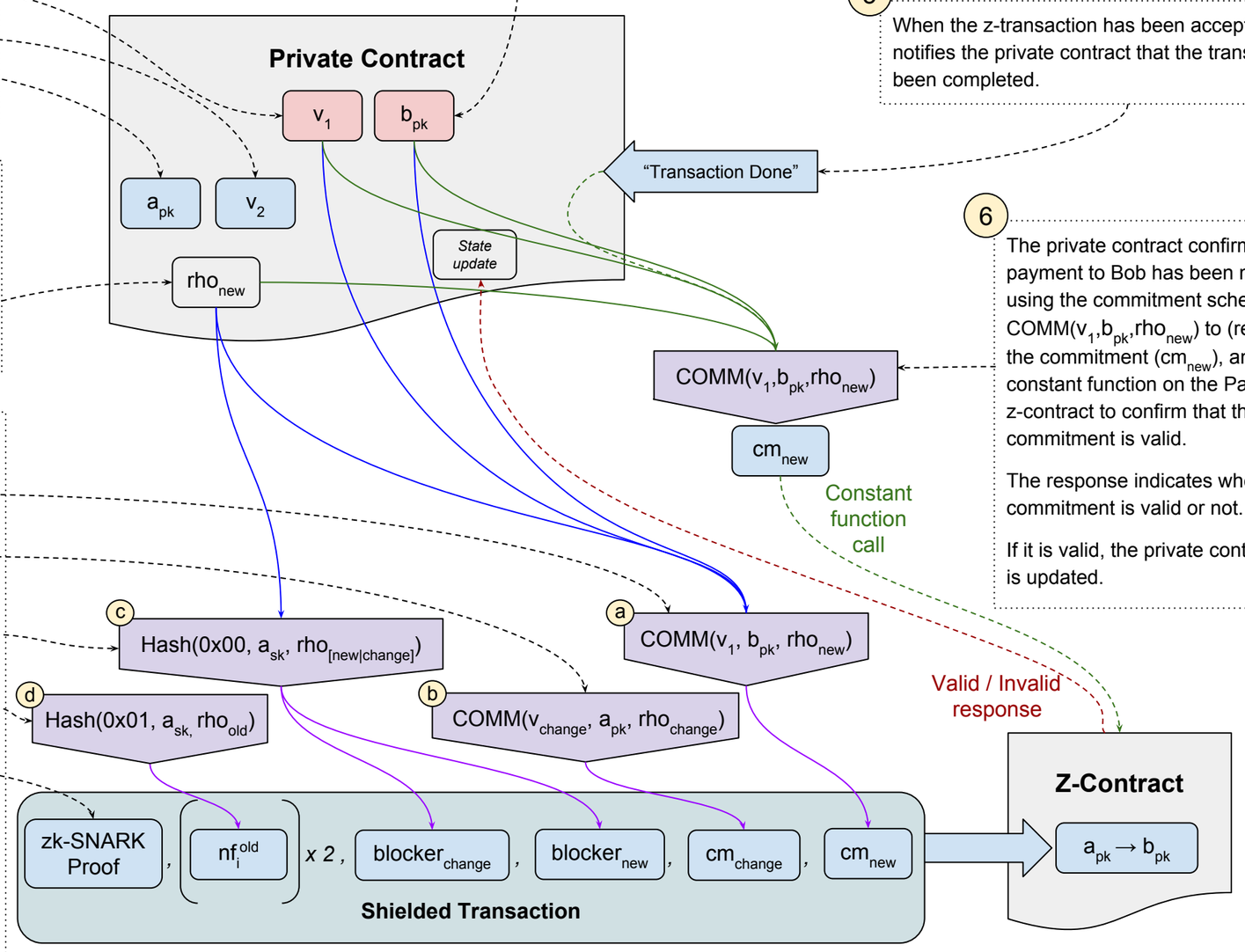
The role of the blocker is to ensure that every note's nullifier is unique.

Alice then creates the z-transaction by calling the relevant function on the Payment z-contract and supplying:

- the zk-SNARK proof,
- the input notes' nullifiers (nf_i^{old})
- the blockers ($blocker_{new}$ and $blocker_{change}$), and
- the new commitments (cm_{new} and cm_{change}).

The Payment z-contract

- Ensures that the blockers are unique,
- ensures that the input notes haven't previously been spent (by confirming that their nullifiers haven't previously been revealed),
- validates the z-transaction by verifying the zk-SNARK proof (using the zk-SNARK transaction verification precompile), and
- records the nullifiers, blockers, and commitments.



5

When the z-transaction has been accepted, Alice notifies the private contract that the transfer has been completed.

6

The private contract confirms that the payment to Bob has been made by using the commitment scheme COMM(v_1,b_{pk},rho_{new}) to (re-)generate the commitment (cm_{new}), and calls a constant function on the Payment z-contract to confirm that the commitment is valid.

The response indicates whether the commitment is valid or not.

If it is valid, the private contract's state is updated.

7-10

Steps 7 thru 10 are similar to steps 3 thru 6, except Bob is making a shielded transaction to send ACME z-tokens to Alice using v_2, a_{pk}, and rho_{new}.

Appendix C: Glossary

Note - The shielded equivalent of an unspent transaction output. The word "note" is also used to describe the data elements that the note's Recipient must possess in order to spend the note (in this document, these are referenced as v, rho and a_{pk}).

Shielded transaction - Transfers a note from one party to another. This is the equivalent of a z-addr->z-addr Zcash transaction. The Sender, Recipient and number of assets being transferred are not visible to a third-party observer.

Shielding transaction - Turns transparent tokens into notes. This is the equivalent of a t-addr->z-addr Zcash transaction. The Sender's payment address and the number of tokens that are being shielded are visible to a third party observer but there is no way to identify the Recipient(s).

Unshielding transaction - Turns notes into transparent tokens. This is the equivalent of a z-addr ->t-addr Zcash transaction. The Recipient's payment address and the number of tokens that are being unshielded are visible to a third party observer but there is no way to identify the Sender.

Z-contract - A smart contract, visible on the main Quorum chain, which implements ZSL, and supports the issuance, shielding, shielded transfer and unshielding of tokens.

Z-transactions - A catch-all term to describe any transaction that involves a zk-SNARK proof. Includes shielded, shielding and unshielded transactions.

ZSL Demo API - A JavaScript API module which can be accessed via the Quorum JavaScript console or via HTTP-RPC.