Atomic Crosschain Transactions

Peter Robinson*†, Raghavendra Ramesh*, John Brainard*, Sandra Johnson*‡
*Protocol Engineering Group and Systems (PegaSys), ConsenSys
†School of Information Technology and Electrical Engineering, University of Queensland, Australia
‡ACEMS, Queensland University of Technology, Australia
peter.robinson@consensys.net, raghavendra.ramesh@consensys.net, johngbrainard@gmail.com,
sandra.johnson@consensys.net
February 25, 2020 (Version 0.8)

***Abstract*—Atomic Crosschain Transaction technology allows composable programming across private Ethereum blockchains. It allows for inter-contract and inter-blockchain function calls that are both synchronous and atomic: if one part fails, the whole call graph of function calls is rolled back. It is not based on existing techniques such as Hash Time Locked Contracts, relay chains, block header transfer, or trusted intermediaries. BLS Threshold Signatures are used to prove to validators on one blockchain that information came from another blockchain and that a majority of the validators of that blockchain agree on the information. Coordination Contracts are used to manage the state of a Crosschain Transaction and as a repository of Blockchain Public Keys. Dynamic code analysis and signed nested transactions are used together with live argument checking to ensure execution only occurs if the execution results in valid state changes. Contract Locking and Lockability enable atomic updates.**

I. Introduction

Atomic Crosschain Transactions [1] for permissioned Ethereum blockchains [2] allow for inter-contract and inter-blockchain function calls that are both synchronous and atomic. Atomic Crosschain Transactions are special nested Ethereum transactions that include additional fields to facilitate the atomic behaviour securely. The technology has been designed to shield application developers from the complexity of crosschain transactions by incorporating the required changes into the Ethereum Client software.

Fig. 1 shows a system of blockchains. Enterprises are indicated by the colour of line between two blockchains. For example, the enterprise represented by the green lines has nodes on blockchains A, C, D, X, and Y; the enterprise represented by the blue lines has nodes on blockchains A, B, D, and X; and the enterprise represented by the red lines has nodes on all blockchains. Using Atomic Crosschain Transaction technology, an enterprise can create a crosschain transaction that starts on any blockchain they have a node on. The call graph of function calls can go across blockchains, using any blockchain that they have a node on. The Coordination Blockchains are used to hold the cross transaction state. The only limitation on crosschain transactions is that all nodes on all blockchains need to be able to access to the Coordination Blockchain used in a crosschain transaction.

Fig. 2 shows code fragments from two contracts on two blockchains. The function `funcA` in contract `ConA` on Private Blockchain A calls the function `funcB` in contract `ConB` on Private Blockchain B. Atomic Crosschain Transactions allow the crosschain function call to occur synchronously and atomically. By synchronously it is meant that function calls that return results can be executed and the results are immediately available and that all execution will complete and updates will be committed prior to the end of the Atomic Crosschain Transaction. By atomic it is meant that the updates are either applied on both blockchains, or the updates are discarded on both blockchains.

Atomic Crosschain Transactions technology does not rely on Time Hash Lock Contracts [3], relay chains [4], block header relaying [5], [6], or trusted intermediaries. It is a new technique which brings together BLS Threshold Signatures, nested transactions, and dynamic parameter checking against expected signed values, and a Coordination Blockchain as a holder of crosschain transaction state and as a time-out time reference.

Section II *Components* describes the concepts behind the technology. Section III *Transaction Processing* walks through happy case and failure case transaction processing. Section IV *Application Development* discusses considerations that should be considered when creating applications with this technology. Section V analyses the Safety and Liveness properties of this technology.

Source code for this technology is available on github.com [7].

II. Components

A. Nested Transactions

Atomic Crosschain Transactions are nested Ethereum transactions and views. Transactions are function calls that update state. Views are function calls that return a value but do

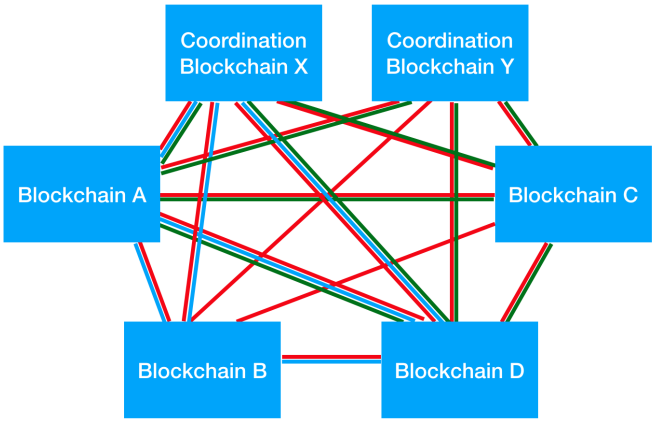

Fig. 1: Cross-Blockchain Linkage

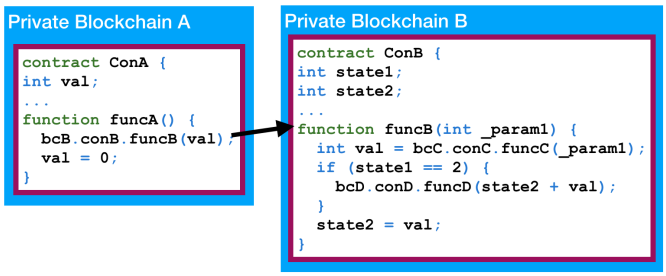

Fig. 2: Function Calls across Blockchains

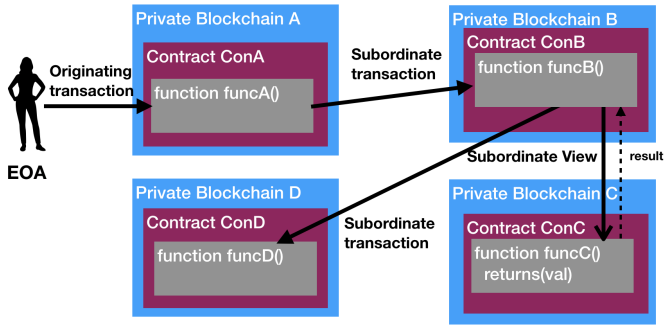

Fig. 3: Originating Transaction containing Two Nested Subordinate Transactions and a Subordinate View

not update state. Fig. 3 shows an Externally Owned Account (EOA) calling a function `funcA` in contract `ConA` on blockchain `Private Blockchain A`. This function in turn calls function `funcB`, that in turn calls functions `funcC` and `funcD`, each on separate blockchains. The transaction submitted by the EOA is called the *Originating Transaction*. The transactions that the Originating Transaction causes to be submitted are called Subordinate Transactions. Subordinate Views may also be triggered. In Fig. 3, a Subordinate View is used to call `funcC`. This function returns a value to `funcB`.

Fig. 4 shows the nested structure of the Atomic Crosschain Transaction. The EOA user first creates the signed Subordinate View for `Private Blockchain C`, contract `ConC`, function `funcC` and the signed Subordinate Transaction for `Private Blockchain D`, contract `ConD`, function `funcD`. They then create the signed Subordinate Transaction for `Private Blockchain B`, contract `ConB`, function `funcB`, and include the signed Subordinate Transaction and View. Finally, they sign the Originating Transaction for `Private Blockchain A`, contract `ConA`, function `funcA`, including the signed Subordinate Transactions and View.

B. Actual and Expected Parameter Values

When a function executes in the Ethereum Virtual Machine [8] function parameter values and stored state combine to form the actual values of variables during execution. For example, consider `funcB` in contract `ConB` on `Private Blockchain B` in Fig. 2. Assuming `_param1` is `1`, `state1` is `2`, `state2` is `4`, and that `funcC` returns the value `6`, then function `funcC` will be called with the parameter value `1`, and function `funcD` will be called with the parameter value `10`. To execute this as part of a Crosschain Transaction, signed Subordinate Transactions and View need to be created with the appropriate parameter values.

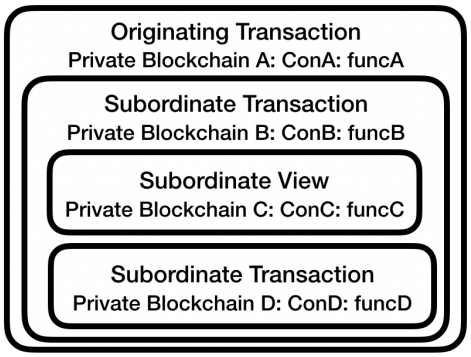

Fig. 4: Nested Transactions and Views

Execution of a transaction or view fails if the actual parameter value passed to a function does not match the value in the signed Subordinate Transaction or View. The parameter value in the signed Subordinate Transaction or View is the value the application developer expected to be passed to the function. The expected value can be determined at time of nested transaction creation using dynamic program analysis techniques. In particular, the dynamic analysis needs to consider program flow. For instance, if `state1` was `1`, then `funcD` would not be called, and no Subordinate Transaction should be included in the Crosschain Transaction for this function call.

C. Per-Node Transaction Processing

When the EOA submits the Originating Transaction to a node, the node processes the transaction using the algorithm shown in Listing 1. If the transaction includes any Subordinate Views, they are dispatched and their results are cached (Lines 1 to 3). The function is then executed (Lines 4 to 17). If a Subordinate Transaction function call is encountered, the node checks that the parameter values passed to the Subordinate Transaction function call match the parameter values in the signed Subordinate Transaction (Lines 6 to 8). If a Subordinate View function call is encountered, the node checks that the parameters passed to the Subordinate View function call match the parameter values in the signed Subordinate View (Lines 9 and 10). The cached values of the results of the Subordinate View function calls are then returned to the executing code (Line 11). If the execution has completed without error, then each of the signed Subordinate Transactions is submitted to a node on the appropriate blockchain (Nodes 18 to 20).

Listing 1: Originating or Subordinate Transaction Processing

```
1  For All Subordinate Views {
2    Dispatch Subordinate Views & cache results
3  }
4  Trial Execution of Function Call {
5    While Executing Code {
6      If Subordinate Transaction function called {
7        check expected & actual parameters match.
8      }
9      Else If Subordinate View function is called {
10       check expected & actual parameters match
11       return cached results to code
12     }
13     Else {
14       Execute Code As Usual
15     }
16   }
17 }
```

```
18 For All Subordinate Transactions {
19   Submit Subordinate Transactions
20 }
```

D. Blockchain Signing and Threshold Signatures

BLS Threshold Signatures [9], [10] combines the ideas of threshold cryptography [11] with Boneh-Lynn-Shacham(BLS) signatures [12], and uses a Pedersen commitment scheme [13] to ensure verifiable secret sharing. The scheme allows any `M` validator nodes of the total `N` validator nodes on a blockchain to sign messages in a distributed way such that the private key shares do not need to be assembled to create a signature. Each validator node creates a signature share by signing the message using their private key share. Any `M` of the total `N` signature shares can be combined to create a valid signature. Importantly, the signature contains no information about which nodes signed, or what the threshold number of signatures (`M`) needed to create the signature is.

The Atomic Crosschain Transaction system uses BLS Threshold Signatures to prove that information came from a specific blockchain. For example, in Fig. 3, nodes on `Private Blockchain B` can be certain of results returned by a node on `Private Blockchain C` for the function call to `funcC`, as the results are threshold signed by the validator nodes on `Private Blockchain C`. Similarly, validator nodes on `Private Blockchain A` can be certain that validator nodes on `Private Blockchain B` have mined the Subordinate Transaction, locked contract `ConB` and are holding the updated state as a provisional update because validator nodes sign a *Subordinate Transaction Ready* message indicating that the Subordinate Transaction is ready to be committed.

E. Multichain Nodes

A Multichain Node is a logical grouping of one or more blockchain validator nodes, where each node is on a different blockchain. The blockchain nodes operate together to allow Crosschain Transactions. The Multichain Node on which the transaction is submitted must have Validator Nodes on all of the blockchains on which the Originating Transaction and Subordinate Transactions and Views take place.

Fig. 5 shows four enterprises that have validator nodes on `Private Blockchain A` to `Private Blockchain D`. Alice who works in Enterprise 1 can submit Atomic Crosschain Transactions that span `Private Blockchain A` to `Private Blockchain D` as Enterprise 1 has a Multichain Node that includes validator nodes on each blockchain. However, Bob who works in Enterprise 4 can only submit Atomic Crosschain Transactions that span `Private Blockchain B` and `Private Blockchain C` as Enterprise 4 only has validator nodes on `Private Blockchain B` and `Private Blockchain C`.

F. Crosschain Coordination

Crosschain Coordination Contracts exist on *Coordination Blockchains*. They allow validator nodes to determine whether the provisional state updates related to the Originating Transaction and Subordinate Transactions should be committed or discarded. The contract is also used to determine a common time-out for all blockchains, and as a repository of Blockchain Public Keys.

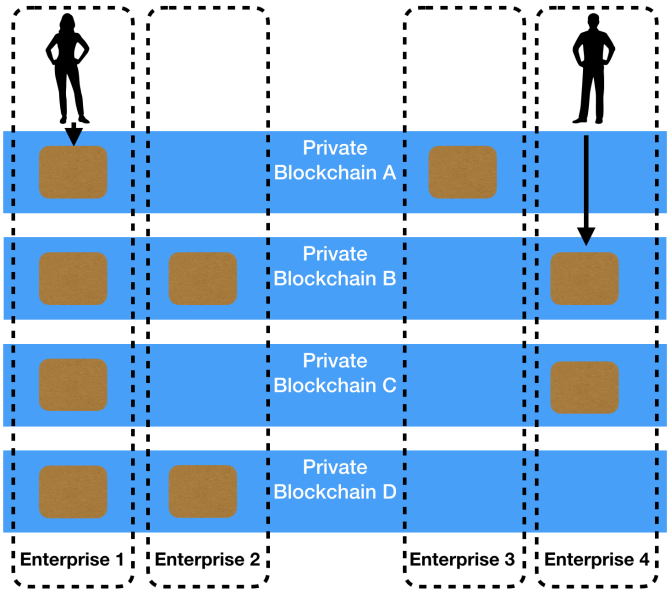

Fig. 5: Multichain Nodes

When a user creates a Crosschain Transaction, they specify the Coordination Blockchain and Crosschain Coordination Contract to be used for the transaction, and the time-out for the transaction in terms of a block number on the Coordination Blockchain. The validator node that they submit the Originating Transaction to (the *Originating Node*) works with other validator nodes on the blockchain to sign a *Crosschain Transaction Start* message. This message is submitted to the Crosschain Coordination Contract to indicate to all nodes on all blockchains that the Crosschain Transaction has commenced.

When the Originating Node has received Subordinate Transaction Ready messages for all Subordinate Transactions, it works with other validator nodes to create a *Crosschain Transaction Commit* message. This message is submitted to the Crosschain Coordination Contract to indicate to all nodes on all blockchains that the Crosschain Transaction has completed and all provisional updates should be committed. If an error is detected, then a *Crosschain Transaction Ignore* message is created and submitted to the Crosschain Coordination Contract to indicate to all nodes on all blockchains that the Crosschain Transaction has failed and all provisional updates should be discarded. Similarly, if the transaction times-out, all provisional updates will be discarded.

G. Contract Locking and Provisional State Updates

When a contract is first deployed it is marked as a Lockable Contract or a Nonlockable Contract. A Nonlockable Contract, the default, is one which can not be locked. When a node attempts to update the state of a contract given an Originating or Subordinate Transaction, it checks whether the contract is *Lockable* and whether it is *locked*. The transaction fails if the contract is Nonlockable or if the contract is Lockable but is locked.

Figure 6 shows the locking state transitions for a contract. The Crosschain Coordination Contract will be in *Started* state. The act of mining an Originating Transaction or Subordinate

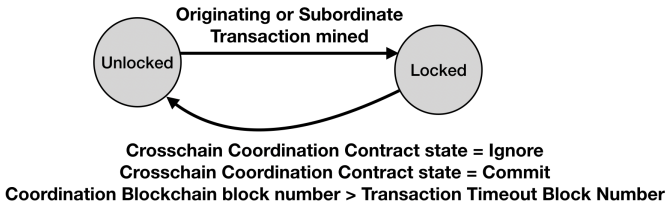

Fig. 6: Contract Locking States

Transaction and including it in a blockchain locks the contract. The contract is unlocked when the Crosschain Coordination Contract is in the *Committed* or *Ignored* state, or when the block number on the Coordination Blockchain is greater than the Transaction Timeout Block Number. The Crosschain Coordination Contract will change from the *Started* state to the *Committed* state when a valid Crosschain Transaction Commit message is submitted to it, and it will change from the *Started* state to the *Ignored* state when a valid Crosschain Transaction Ignore message is submitted to it.

Ordinarily, all nodes will receive a message indicating that they should check the Crosschain Coordination Contract when the contract can be unlocked. When a node first processes a transaction, it will set a local timer which should expire when the Transaction Timeout Block Number is exceeded. If the node has not received the message by the time the local timer expires, the node checks the Crosschain Coordination Contract to see if the Transaction Timeout Block Number has been exceeded and whether the updates should be committed or ignored.

H. Crosschain Transaction Fields

Originating Transactions, Subordinate Transactions, and Subordinate Views contain the fields shown in Table I. Some of the information in the standard Ethereum transaction fields are exposed to blockchain application contract code, such as the `value` field via the Solidity code `msg.value`. The new extended crosschain transaction fields are made available to blockchain application contract code via a precompile contract.

All nodes that process the transaction check that the `Coordination Blockchain Id`, `Crosschain Coordination Contract`, `Crosschain Transaction Time-out`, `Crosschain Transaction Id`, and `Originating Blockchain Id` are consistent across the transaction or view they are processing, and the nested Subordinate Transactions and Views. The nodes also check that the `To` address and `From Address`, and the blockchain identifier obtained from the `V` field and the `From Blockchain Id` match across transactions and views.

The `To` address is the address of the contract containing the function called on a blockchain. For example, the function (f1) in contract (c1) could call a function (f2) in another contract (c2) on the same blockchain (b1). The second contract (c2) could call a function (f3) in a contract (c3) on another blockchain (b2) via a Subordinate Transaction. The `From Address` of the Subordinate Transaction will match the `To` address of the transaction on the first blockchain (b1). This will be the address first contract (c1). It will however, not match the address of the second contract (c2), which is the function that caused the Subordinate Transaction to be triggered.

Field	Description
Standard Ethereum Transaction Fields	
Nonce	Per-account, per-blockchain transaction number.
GasPrice	Amount offered to pay for gas for the transaction.
GasLimit	Maximum gas which can be used by the transaction.
To	Address of the account to send the value to, or the address of a contract to call.
Value	Amount of Ether to transfer.
Data	Encoded function signature and parameter values.
V	Part of the transaction digital signature & blockchain identifier this transaction must execute on.
R	Part of the transaction digital signature.
S	Part of the transaction digital signature.
Additional Crosschain Transaction Fields	
Type	Type of crosschain transaction (e.g. Originating Transaction)
Coordination Blockchain Id	Blockchain identifier of Coordination Blockchain to use for this transaction.
Crosschain Coordination Contract	Address of the Crosschain Coordination Contract to use for this transaction.
Crosschain Transaction Time-out	Coordination Blockchain block number when this transaction will time out.
Crosschain Transaction Id	Identifies this crosschain transaction.
Originating Blockchain Id	Blockchain identifier of the blockchain the Originating Node is on.
From Blockchain Id	Blockchain identifier of the blockchain that the function call executed on that resulted in this Subordinate Transaction or View being submitted.
From Address	*To* address from the transaction or view that resulted in this Subordinate Transaction or View.
Subordinates	List of Subordinate Transactions and Subordinate Views that are called directly from this transaction or view.

TABLE I: Crosschain Transaction Fields

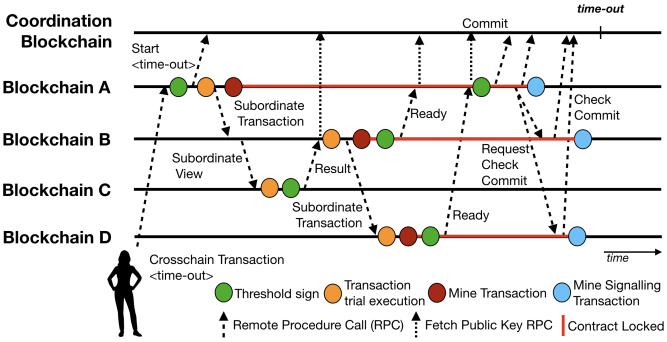

Fig. 7: Sequence Diagram

III. Transaction Processing

A. Happy Case

Fig. 7 shows a sequence diagram showing an example sequence of execution, based on the calls in Fig. 3. The sequence starts with an Externally Owned Account (EOA) submitting a Crosschain Transaction to a node on Blockchain A. This nested transaction contains the Originating Transaction, and all Subordinate Transactions and Views, plus a time-out in terms of Coordination Blockchain block numbers. The node is known as the Coordinating Node for this transaction.

The nodes on Blockchain A cooperate to threshold sign the *Start* message. The signed *Start* message is submitted to the Coordination Contract on the Coordination Blockchain specified in the Originating Transaction (and all Subordinate Transactions and Views). The Coordination Contract on the Coordination Blockchain accepts the *Start* message if its

signature can be verified with the public key registered for Blockchain A. The *Start* message contains the time-out block number, which the Coordinating Contract registers when it accepts the *Start* message. Once the *Start* is registered, the Crosschain Transaction is in *Start* state.

The Coordinating Node then *Trial Executes* the Originating Transaction. Assuming the code executes correctly, the Subordinate Transaction for Blockchain B is dispatched. The Originating Transaction is then submitted to Blockchain A's transaction pool. When the transaction is included in a block, the contract's updated state is stored as provisional state, and the contract state is locked.

The Subordinate Transaction for Blockchain B contains a Subordinate View for Blockchain C. The Subordinate View is dispatched to a node on Blockchain C. The node on Blockchain C completes a *Trial Execution* of the Subordinate View at a certain block number. The node then cooperates with other Blockchain C nodes to have the result of the Subordinate View at the specific block number threshold signed. The node returns the signed result to the calling node on Blockchain B.

The nodes on Blockchain B that do not contain the correct version of Blockchain C's public key fetch it from the Coordination Contract. The node *Trial Executes* the Subordinate Transaction, returning the value from the Subordinate View to the code when the function corresponding to the Subordinate View is called. Assuming the *Trial Execution* executes correctly, then the Subordinate Transaction to be executed on Blockchain D is dispatched to a node on Blockchain D. The Subordinate Transaction for Blockchain B is submitted to Blockchain B's transaction pool, with the signed Subordinate View result from Blockchain C attached. When the transaction is included in a block, the contract's updated state is stored as provisional state, and the contract state is locked. The nodes on Blockchain B then cooperate to threshold sign a *Ready* message, indicating that the blockchain is ready to commit the Atomic Crosschain Transaction. The message is submitted to the Coordinating Node.

The Subordinate Transaction for Blockchain D is trial executed. There are no Subordinate Transactions or Views for the transaction that need to be dispatched. The transaction is mined. When the transaction has been included in a block, the updated state is stored in provisional state, the contract is locked, and the nodes on Blockchain D cooperate to threshold sign a *Ready* message, indicating that the blockchain is ready to commit the Atomic Crosschain Transaction. The message is submitted to the Coordinating Node.

When the Coordinating Node (on Blockchain A) has received *Ready* messages for Blockchain B and D, and the Originating Transaction has been mined, then the nodes on Blockchain A then cooperate to threshold sign a *Commit* message. This message is uploaded to the Coordination Contract on the Coordination Blockchain. Assuming this message is uploaded prior to the time-out, the Crosschain Transaction is then in the commit state.

The Coordinating Node requests nodes on other Blockchains commit a Signalling Transaction. When a node on a blockchain encounters a Signalling Transaction for a certain Crosschain Transaction, it checks the Coordination Contract for the Crosschain Transaction to check to see if the transaction should be committed or ignored. If the transaction is to be committed, then all contract provisional state related to the transaction is converted to normal state, unlocking the contracts. If the transaction is to be ignored, then the contract provisional state is discarded and the contract state is updated, unlocking the contracts.

B. Failure Cases

If an error is detected then nodes on Blockchain A can cooperate to create a threshold signed *Ignore* message which is uploaded to the Coordination Contract to indicate to all nodes that the Crosschain Transaction should be abandoned, all provisional state updates should be discarded and contracts should be unlocked. If an *Ignore* message can not be created, or if the Crosschain Transaction times-out, then the state returned by the Coordination Contract indicates the Crosschain Transaction should be abandoned.

When nodes on each blockchain first become a part of a Crosschain Transaction, they set a timer to expire when they expect the Crosschain Transaction to time-out, plus a random additional wait period. The timer is cancelled when a Signalling Transaction is received. If a node's timer expires prior to receiving a Signalling Transaction, the node checks the Coordination Contract and then creates and submits a Signalling Transaction to indicate to other nodes that the updates should be committed or ignored, and that the contract should be unlocked. The additional random wait period is used so that all nodes on the blockchain don't simultaneously send Signalling Transactions.

IV. Application Development

A. Design for Locking

Contracts that have state updated as part of a crosschain transaction are locked. Calls to those locked contracts will fail until they are unlocked. Traditional contracts that hold registries of information, which may need to be accessed by many users at the same time, such as ERC 20 contracts, need to be refactored to work successfully in a crosschain scenario. They should be restructured such that there is a non-lockable *router* contract that determines the appropriate lockable *item* contract.

Fig 8 shows how an ERC 20 contract could be refactored. Each account could have one or more lockable ERC 20 Payment Slot contracts. The non-lockable ERC 20 Router contract's code could be written such that transfers are sent from an account's payment slot contract that isn't locked which has adequate funds to a payment slot for the destination account that is not currently locked. In this way, multiple payment may be executed in parallel.

B. Application Level Authentication

As with traditional Ethereum transactions, the type of application level authentication required for a Crosschain Transaction will be application dependent.

1) No Authentication: Many functions will need no authentication at all. That is, functions can be designed such that it is safe to execute a transaction or return results of a view to any caller who is able to access the function.

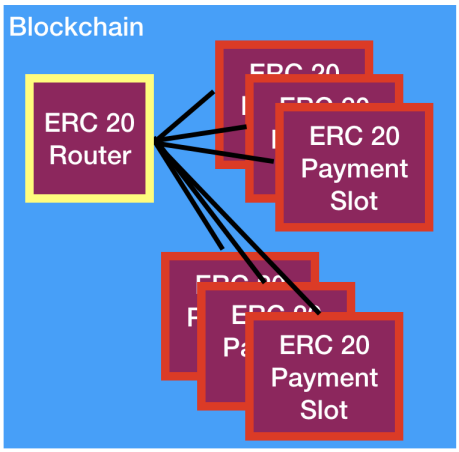

Fig. 8: ERC 20 Contract for Crosschain

2) Using `msg.sender` or `tx.origin`: From the perspective of each Originating Transaction, Subordinate Transaction or View, `msg.sender` and `tx.origin` operate in the same way as a standard Ethereum transaction. That is, if an EOA submitted a transaction that called a function in contract A that then called a function in contract B on the same blockchain, `msg.sender` for contract B is contract A, and is the EOA for contract A. In both cases `tx.origin` would be the EOA. In the context of a node processing an Originating Transaction, Subordinate Transaction or View, for the purposes of `msg.sender` and `tx.origin`, the transaction or view appears as a separately signed transaction. Given the similarities with standard Ethereum, `msg.sender` and `tx.origin` could be used in the same way as standard Ethereum to authenticate which EOA or contract on the same blockchain called a function call using code similar to that shown in Listing 2.

Listing 2: Checking msg.sender

```
function receiver() external {
  require(msg.sender == authorisedAddress);
  ...
}
```

A key difference between standard Ethereum views and Subordinate Views is that Subordinate Views are signed. As such, the variables `msg.sender` and `tx.origin` can be used within Subordinate Views, whereas they are not set in the context of normal Ethereum views (except for the case of `msg.sender` when one contract calls another contract).

3) From Blockchain Id, From Address, and Originating Blockchain Id: If a contract needs to only respond to calls from a certain contract on a certain blockchain, then the code in Listing 3 should be used. The code checks that the `From Blockchain Id` and `From Address` match the authorised blockchain and address, and checks that the blockchains represented by `From Blockchain Id` and `Originating Blockchain Id` are semi-trusted. By semi-trusted it is meant that fewer than `M` validators operating the blockchain are Byzantine. Note that this scenario implies the contract should allow for any `msg.sender` and `tx.origin`.

Listing 3: Crosschain Application Authentication

```
function receiver() external {
  address fromAddr = infoPrecompile(FROM_ADDR);
  uint256 fromBcId = infoPrecompile(FROM_BCID);
  uint256 origBcId = infoPrecompile(ORIG_BCID);
  require(fromAddr == authorisedFromAddress);
  require(fromBcId == authorisedFromBcId);
  require(origBcId == authorisedOrigBcId);
  ...
}
```

V. Analysis

Typically, any desired property of a distributed system can be expressed as a combination of safety and liveness properties. Treading on the same lines, atomicity property of the atomic crosschain protocol can also be seen as a combination of safety and liveness properties. In general, safety property captures the notion that bad states are not reachable, and liveness property captures the notion that good states are eventually reached. For atomicity, the safety property translates to: when the protocol finishes, the whole crosschain transaction gets either committed or rolled back, the liveness property states that: the atomic crosschain transaction protocol eventually (after finite amount of time) terminates. We present a more formal discussion here.

Limit on the number of Byzantine nodes. Consensus protocols invariably has a limit on the number of Byzantine nodes in a network. For example, proof-of-work protocol requires the number of Byzantine nodes to be less than half of the total number of nodes. Similarly, IBFT and IBFT2 protocols has a limit of one-third the total number of nodes. Because our atomic crosschain protocol is built on top of consensus protocols, we require and **assume** that the conditions for the correct operation of the consensus protocols are met. In other words, we inherit the same limit on the number of Byzantine nodes from the dependant consensus protocol.

Theorem 1 (Safety)**.** *Suppose that the atomic crosschain protocol has finished executing a crosschain transaction c. Then:*

- *if c succeeds then the protocol successfully commits the state updates of all the associated (originating and subordinate) transactions,*
- *if c fails then the protocol rolls back the state updates of all the associated transactions.*

Proof. We prove by case analysis of all possible states of the crosschain transaction c recorded in the coordination contract.

- **not-started** This is possible only if c fails before the state of the coordination contract is updated to *started*. At this point none of the originating transaction or the subordinate transactions have started to execute. Hence there is nothing to rollback.
- **started** We show that this situation is not possible. The coordination contract has received the *start* message from the originating chain, but not the *commit* or *ignore* message. Based on the time-out specified in the *start* message, the nodes on the originating chain start their respective local timers, after processing the originating transaction. Similarly, the nodes on the subordinate chains start local timers after processing their respective subordinate transactions. After timers expire, the nodes look up the state of c in the coordination contract. In case of any failure in processing the originating transaction or in one of the subordinate transactions, the originating

chain would have updated the coordination contract to *ignored* state. Also, the state of the coordination contract is updated to *ignored* state when there is a time-out. So, when c is processed (either successfully or not) the state of c in coordination contract can only end up in *committed* or *ignored*, after having been in the *started* state.

- **committed** It means that the coordination contract received the threshold signed *commit* message from the originating chain. After submitting the *commit* message, originating sidechain's coordinating node messages all the validators in its chain and other coordinating nodes on the subordinate transaction sidechains to check the state of coordination contract. If it is Byzantine, then anyways the local timers on the nodes (of all participating chains) expire and the nodes look up the coordination contract for the state of c, and find that c is *committed*. They then commit their local provisional states. Hence the whole transaction c is committed.
- **ignored** This is possible when either the coordinating node on the originating chain has sent an *ignore* message, or c has timed out, and the coordination contract updates the state of c to be *ignored*. Nodes will commit or roll back their provisional states only after looking up the coordination contract. In this case all nodes on all participating chains roll back their updates, implying that the whole crosschain transaction c is rolled back.

□

Theorem 2 (Liveness)**.** *Suppose the atomic crosschain protocol is started, meaning, a crosschain transaction is submitted to a coordinating node on the originating sidechain. Then, the atomic crosschain protocol protocol will eventually (after finitely many steps) finish.*

Proof. The intuition of the proof is simple. The protocol itself has a finite number of steps. Also, as the *start* message is submitted, all validator nodes on all participating chains start their local timer. This ensures that the whole transaction to time out eventually, in case there are indefinite delays.

We consider the special case when the crashing (or becoming Byzantine) of nodes might make the protocol not to terminate. The crashing of other nodes are governed by the dependant consensus protocol. Hence we assume that we always have a minimum number of nodes to be non-Byzantine as dictated by the consensus protocol. We examine if crashing a coordinating node introduces indefinite waits, which implies that the protocol does not terminate.

- Suppose the coordinating node on originating chain crashes before submitting crosschain transaction *start* message to the coordination contract. Then no other chain knows about this crosschain transaction and this transaction is terminated and deemed failed.
- Suppose the coordinating node on originating chain crashes after submitting *start* message but before submitting the *commit* message, then the whole transaction times out. The local timers on validator nodes of originating sidechain time out, look up the coordination contract, find the transaction is *ignored*, and abort the transaction. The subordinate chains complete the execution of views/-transactions and send their subordinate transaction *ready* messages to the coordinating node of the originating chain. Because the coordinating node of originating chain has crashed, they observe that the state of the coordinating contract is never changed to *committed*, and the validator nodes on all participating chains will eventually time out, look up the coordination contract (which is set to *ignored* after time out), and abort the transaction.
- Suppose the coordinating node on originating chain crashes after submitting the crosschain transaction *commit* message to the coordination contract. Then (validator nodes on) all chains see the *committed* state of coordinating contract and will commit the transaction.
- Suppose the coordinating node on a subordinate chain crashes. Then it cannot send the subordinate transaction *ready* message to the originating chain. The coordinating node on the originating chain keeps waiting and times out. The coordination contract gets updated to *ignored* state and the transaction terminates and is rolled back.

□

Establishing the liveness property is very useful as it implies the properties of termination, deadlock-free and livelock-free.

VI. Conclusion

Atomic Crosschain Transactions allow application developers to create complex cross-blockchain applications in a straightforward manner. It does this by absorbing the vast majority of the complexity required to deliver this behaviour using a novel combination of technologies.

Acknowledgment

This research has been undertaken whilst we have been employed full-time at ConsenSys. Peter acknowledges the support of University of Queensland where he is completing his PhD, and in particular the support of my PhD supervisor Dr Marius Portmann.

We acknowledge the co-authors of the original Atomic Crosschain Transaction paper [1] Dr David Hyland-Wood and Roberto Saltini for their help creating the technology upon which this paper is based. We thank Dr Catherine Jones and Horacio Mijail Anton Quiles for reviewing this paper and providing astute feedback.

References

[1] P. Robinson, D. Hyland-Wood, R. Saltini, S. Johnson, and J. Brainard, "Atomic Crosschain Transactions for Ethereum Private Sidechains," 2019. [Online]. Available: https://arxiv.org/abs/1904.12079

[2] P. Robinson, "Requirements for Ethereum Private Sidechains," 2018. [Online]. Available: http://adsabs.harvard.edu/abs/2018arXiv180609834R

[3] "Hashed Timelock Contracts," 2017. [Online]. Available: https://en.bitcoin.it/wiki/Hashed_Timelock_Contracts

[4] E. Buchman and J. Kwon, "Cosmos: A network of distributed ledgers." Github, 2016. [Online]. Available: https://github.com/cosmos/cosmos/blob/master/WHITEPAPER.md

[5] Clearmatics, "General interoperability framework for trustless cross-system interaction," 2018. [Online]. Available: https://github.com/clearmatics/ion

[6] J. Chow, "BTC Relay," 2016. [Online]. Available: https://media.readthedocs.org/pdf/btc-relay/latest/btc-relay.pdf

[7] PegaSys, "Atomic Crosschain Transaction Sample Code Github Repository." [Online]. Available: https://github.com/PegaSysEng/sidechains-samples

[8] G. Wood, "Ethereum: a secure decentralized generalised transaction ledger," Github, p. Github site to create pdf, 2016. [Online]. Available: https://github.com/ethereum/yellowpaper

[9] A. Boldyreva, "Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme." [Online]. Available: http://www-cse.ucsd.edu/users/aboldyre

[10] P. Robinson, “Ethereum Engineering Group: BLS Threshold Signatures - YouTube.” [Online]. Available: https://www.youtube.com/watch?v=XZTvBYG9pn4
[11] A. Shamir, “How to share a secret,” *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979. [Online]. Available: https://cs.jhu.edu/ sdoshi/crypto/papers/shamirturing.pdf
[12] D. Boneh, B. Lynn, and H. Shacham, “Short Signatures from the Weil Pairing,” *Journal of Cryptology*, vol. 17, no. 4, pp. 297–319, 2004. [Online]. Available: https://doi.org/10.1007/s00145-004-0314-9
[13] T. P. Pedersen and D. W. Davies, “A Threshold Cryptosystem without a Trusted Party,” in *Advances in Cryptology EUROCRYPT ’91*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 522–526. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-46416-6_47
[14] Y.-T. Lin, “EIP 650: Istanbul Byzantine Fault Tolerance,” 2017. [Online]. Available: https://github.com/ethereum/EIPs/issues/650
[15] R. Saltini, “IBFT 2.0 Gray Paper version 0.2,” 2019. [Online]. Available: https://github.com/PegaSysEng/ibft2.x/raw/master/gray_paper/ibft2_gray_paper.pdf
[16] P. Robinson, “Ethereum Engineering Group: Advanced Solidity and Design Patterns - YouTube.” [Online]. Available: https://www.youtube.com/watch?v=VhzafmGGmzo&t=534s