Atomic Commitment Across Blockchains

Victor Zakhary
UC Santa Barbara
Santa Barbara, California
USA, 93106
victorzakhary@ucsb.edu

Divyakant Agrawal
UC Santa Barbara
Santa Barbara, California
USA, 93106
divyagrawal@ucsb.edu

Amr El Abbadi
UC Santa Barbara
Santa Barbara, California
USA, 93106
elabbadi@ucsb.edu

ABSTRACT

The recent adoption of blockchain technologies and open permissionless networks suggest the importance of peer-to-peer atomic cross-chain transaction protocols. Users should be able to atomically exchange tokens and assets without depending on centralized intermediaries such as exchanges. Recent peer-to-peer atomic cross-chain swap protocols use hashlocks and timelocks to ensure that participants comply to the protocol. However, an expired timelock could lead to a violation of the all-or-nothing atomicity property. An honest participant who fails to execute a smart contract on time due to a crash failure, denial of service attacks or even network delays might end up losing assets. Although a crashed participant is the only participant who ends up worse off, current proposals are unsuitable for atomic cross-chain transactions in asynchronous environments where crash failures and network delays are the norm. In this paper, we present AC^{3}WN, the first decentralized all-or-nothing *atomic* cross-chain commitment protocol. The redeem and refund events of the smart contracts that exchange assets are modeled as conflicting events. An open permissionless network of witnesses is used to guarantee that conflicting events could never simultaneously occur and either all smart contracts in an atomic cross-chain transaction are redeemed or all of them are refunded.

PVLDB Reference Format:
Victor Zakhary, Divyakant Agrawal, Amr El Abbadi. Atomic Commitment Across Blockchains. *PVLDB*, 13(9): 1319 - 1331, 2020.
DOI: https://doi.org/10.14778/3397230.3397231

1 Introduction

The wide adoption of permissionless open blockchain networks by both industry (e.g., Bitcoin [22], Ethereum [28], etc) and academia (e.g., Bzycoin [18], Elastico [19], BitcoinNG [11], Algorand [21], etc.) suggests the importance of developing protocols and infrastructures that support peer-to-peer atomic cross-chain transactions. Users, who usually do not trust each other, should be able to directly exchange their tokens and assets that are stored on different blockchains (e.g., Bitcoin and Ethereum) without depending on trusted third party intermediaries. Decentralized permissionless [20] blockchain ecosystems require infrastructure enablers and protocols that allow users to atomically exchange tokens without giving up trust-free decentralization, the main reasons behind using permissionless blockchains. We motivate the problem of atomic cross-chain transactions and discuss the current available solutions and their limitations through the following example.

Suppose Alice owns X bitcoins and she wants to exchange them for Y ethers. Luckily, Bob owns ether and he is willing to exchange his Y ethers for X bitcoins. to atomically exchange assets that reside in different blockchains. In addition, both Alice and Bob **do not trust** each other and in many scenarios, they might not be co-located to do this atomic exchange in person. Current infrastructures do not support these direct peer-to-peer transactions. Instead, both Alice and Bob need to **independently** exchange their tokens through a trusted centralized exchange, Trent (e.g., Coinbase [3] and Robinhood [4]) either through fiat currency or directly. Using Fiat, both Alice and Bob first exchange their tokens with Trent for a fiat currency (e.g., USD) and then use the earned fiat currency to buy the other token also from Trent or from another trusted exchange. Alternatively, some exchanges (e.g., Coinbase) allow their customers to directly exchange tokens (e.g., ether for bitcoin or bitcoin for ether) without going through fiat currencies.

These solutions have many drawbacks that make them unacceptable solutions for peer-to-peer atomic cross-chain transactions. *First*, they require both Alice and Bob to trust Trent. This centralized trust requirement risks to derail the whole idea of blockchain's trust-free decentralization [22]. *Second*, they require Trent to trade in all involved resources (e.g., bitcoin and ether). This requirement is unrealistic especially if Alice and Bob want to exchange commodity resources (e.g., transfer a car ownership for bitcoin assuming car titles are stored in a blockchain [16]). *Third*, these solutions do not ensure the atomic execution of the transaction among the involved participants. Alice might trade her bitcoin directly for ether or through a fiat currency while Bob has no obligation to execute his part of the swap. Finally, these solutions significantly increase the number of required transactions to achieve the intended cross-chain transaction, and hence drastically increases the imposed fees. One cross-

Proceedings of the VLDB Endowment, Vol. 13, No. 9
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3397230.3397231

chain transaction between Alice and Bob results in either four transactions (two between Alice and Trent and two between Bob and Trent) if fiat is used or at best two transactions (one between Alice and Trent and one between Bob and Trent) if assets are directly swapped.

An **A**tomic **C**ross-**C**hain **T**ransaction, AC^{2}T, is a distributed transaction that spans multiple blockchains. This distributed transaction consists of sub-transactions and each sub-transaction is executed on a blockchain. An **A**tomic **C**ross-**C**hain **C**ommitment, AC^3, protocol is required to execute AC^{2}Ts. This protocol is a variation of traditional distributed atomic commitment protocols (e.g., 2PC [8, 14]). This protocol should guarantee both *atomicity* and *commitment* of AC^{2}Ts. **Atomicity** ensures the **all-or-nothing** property where either all sub-transactions take place or none of them do. **Commitment** guarantees that any changes caused by a cross-chain transaction must eventually take place if the transaction is decided to commit. Unlike in 2PC and other traditional distributed atomic commitment protocols, atomic cross-chain commitment protocols are also trust-free and therefore must **tolerate** maliciousness [16].

A two-party atomic cross-chain commitment protocol was originally proposed by Nolan [1, 23] and generalized by Herlihy [16] to process multi-party atomic cross-chain transactions, or swaps. Both Nolan's protocol and its generalization by Herlihy use smart contracts, hashlocks, and timelocks to execute atomic cross-chain transactions. A smart contract is a self executing contract (or a program) that gets executed in a blockchain once all the terms of the contract are satisfied. A hashlock is a cryptographic one-way hash function $h = H(s)$ that locks assets in a smart contract until a hash secret s is provided. A timelock is a time bounded lock that triggers the execution of a smart contract function after a pre-specified time period.

The atomic swap between Alice and Bob, explained in the earlier example, is executed using Nolan's protocol as follows. Let a participant be the leader of the swap, say Alice. Alice creates a secret s, only known to Alice, and a hashlock $h = H(s)$. Alice uses h to lock X bitcoins in a smart contract SC_1 and publishes SC_1 in the Bitcoin network. SC_1 transfers X bitcoins to Bob if Bob provides the secret s to SC_1 where $h = H(s)$. In addition, SC_1 is locked with a timelock t_1 that refunds the X bitcoins to Alice if Bob fails to provide s to SC_1 before t_1 expires. As SC_1 is published in the Bitcoin network and made public to everyone, Bob can verify that SC_1 indeed transfers X bitcoins to his public address if he provides s to SC_1. In addition, Bob learns h from SC_1. Using h, Bob publishes a smart contract SC_2 in the Ethereum network that locks Y ethers in SC_2 using h. SC_2 transfers Y ethers to Alice if Alice provides the secret s to SC_2. In addition, SC_2 is locked with a timelock $t_2 < t_1$ that refunds the Y ethers to Bob if Alice fails to provide s to SC_2 before t_2 expires.

Now, if Alice wants to redeem her Y ethers from SC_2, Alice must reveal s to SC_2 before t_2 expires. Once s is provided to SC_2, Alice redeems the Y ethers and s gets revealed to Bob. Now, Bob can use s to redeem his X bitcoins from SC_1 before t_1 expires. Notice that $t_1 > t_2$ is a necessary condition to ensure that Bob has enough time to redeem his X bitcoins from SC_1 after Alice provides s to SC_2 and before t_1 expires. If Bob provides s to SC_1 before t_1 expires, Bob successfully redeems his X bitcoins and the atomic swap is marked completed.

The case against the current proposals: If Bob fails to provide s to SC_1 before t_1 expires due to a crash failure, a network partitioning, or a network denial of service at Bob's site, Bob loses his X bitcoins and SC_1 refunds the X bitcoins to Alice. This violation of the atomicity property of the protocol penalizes Bob for a failure that happens out of his control. Although a crashed participant is the only participant who ends up being worse off (Bob in this example), this protocol does not guarantee the atomicity of AC^{2}Ts in asynchronous environments where crash failures, network partitioning, and message delays are the norm.

Another important drawback in Nolan's and Herlihy's protocols is the requirement to sequentially publish the smart contracts in an atomic swap before the leader (Alice in our example) reveals the secret s. This requirement is necessary to ensure that the publishing events of all the smart contracts in the atomic swap *happen before* the redemption of any of the smart contracts. This causality requirement ensures that any malicious participant who declines to publish their payment smart contract cannot take advantage of the protocol. However, the sequential publishing of smart contracts, especially in atomic swaps that include many participants, proportionally increases the latency of the swap to the number of sequentially published contracts.

In this paper, we propose **AC^{3}WN**, the first decentralized all-or-nothing **A**tomic **C**ross-**C**hain **C**ommitment protocol that uses an open **W**itness **N**etwork to coordinate AC^2Ts. The redemption and the refund events of smart contracts in AC^{2}T are modeled as conflicting events. A decentralized open network of witnesses is used to guarantee that conflicting events must never simultaneously take place and either all smart contracts in an AC^{2}T are redeemed or all of them are refunded. Unlike in Nolan's and Herlihy's protocols, AC^{3}WN allows all participants to concurrently publish their contracts in a swap resulting in a drastic decrease in the latency of atomic swaps. Our contributions are summarized as follows:

- We present AC^{3}WN, the first all-or-nothing atomic cross-chain commitment protocol. AC^{3}WN is decentralized and its correctness does not depend on any trusted centralized intermediary.

- We prove the correctness of AC^{3}WN showing that AC^{3}WN achieves both atomicity and commitment of AC^{2}Ts.

- Finally, we analytically evaluate AC^{3}WN in comparison to Herlihy's [16] protocol. Unlike in Herlihy's protocol where the latency of an atomic swap proportionally increases as the number of the sequentially published smart contracts in the atomic swap increases, our analysis shows that the latency of an atomic swap in AC^{3}WN is constant irrespective of the number of smart contracts involved.

The rest of the paper is organized as follows. In Section 2, we discuss the open blockchain data and transactional models. Section 3 explains the cross-chain distributed transaction model and Section 4 presents AC^{3}WN, our atomic cross-chain commitment protocol. The AC^{3}WN protocol is analyzed in Section 5. The protocol is evaluated in Section 6 and the paper is concluded in Section 7.

2 Open Blockchain Models

2.1 Architecture Overview

An open permissionless blockchain system [20] (e.g., Bitcoin and Ethereum) typically consists of two layers: a storage layer and an application layer. **The storage layer** comprises a decentralized distributed ledger managed by an open network of computing nodes. A blockchain system is permissionless if computing nodes can join or leave the network of its storage layer at any moment without obtaining permission from a centralized authority. Each computing node, also called a *miner*, maintains a copy of the ledger. The ledger is a tamper-proof chain of blocks, hence named *blockchain*. Each block contains a set of valid transactions that transfer assets among end-users. **The application layer** comprises end-users who communicate with the storage layer via *message passing* through a client library. End-users have identities, defined by their public keys, and signatures, generated using their private keys. Digital signatures are the end-users' way to generate transactions as explained later in Section 2.3. End-users submit their transactions to the storage layer through a client library. Transactions are used to transfer assets from one identity to another. End-users multicast their transaction messages to mining nodes in the storage layer.

A mining node validates the transactions it receives and valid transactions are added to the current block of a mining node. Miners run a consensus protocol through mining to agree on the next block to be added to the chain. A miner who mines a block gets the right to add the mined block to the chain and multicasts it to other miners. To make progress, miners accept the first received mined block after verifying it and start mining the next block[1]. Sections 2.2 and 2.3 explain the data model and the transactional model of open blockchain systems respectively.

2.2 Data Model

The storage layer stores the ownership information of assets in the system in the blockchain. The ownership is determined through identities and identities are typically implemented using public keys. In addition, the blockchain stores transactions that transfer the ownership of an asset from one identity to another. Therefore, an asset can be tracked from its registration in the blockchain, the first owner, to its last owner in the blockchain. For example, the Bitcoin blockchain stores the information of the most recent owner of every bitcoin in the Bitcoin blockchain. A bitcoin that is linked to Alice's public key is owned by Alice. Also, new bitcoins are generated and registered in the Bitcoin blockchain through mining. Asset ownership transfers are implemented through transactions.

2.3 Transaction Model

A transaction is a digital signature that transfers the ownership of assets from one identity to another. End-users, in the application layer, use their private keys [26] to digitally sign assets linked to their identity to transfer these assets to other identities, identified by their public keys. These digital signatures are submitted to the storage layer via message passing through a client library. It is the responsibility of the miners to validate that end-users can transact only on their own assets. If an end-user digitally signs an asset that is not owned by this end-user, the resulting transaction is not valid and hence rejected by the miners. In addition, miners validate that an asset cannot be spent twice and hence prevent double spending of assets.

Another way to perform transactions in blockchain systems is through **smart contracts**. A smart contract is a program written in some scripting language (e.g., Solidity for Ethereum smart contracts [5]) that allows general program executions by a blockchain's mining nodes. End-users publish a smart contract in a blockchain through a deployment message, *msg*, that is sent to the mining nodes in the storage layer. A deployment message includes the smart contract code in addition to some implicit parameters that are accessible to the smart contract code once the smart contract is deployed. These parameters include the sender's public key, accessed through *msg.sender*, and an optional asset value, accessed through *msg.val*. This optional asset value allows end-users to send some of their blockchain assets to a deployed smart contract. Like transactions, a smart contract is published in a blockchain if it is included in a mined block in this blockchain. We adopt Herlihy's notion of smart contracts as classes in object oriented programming languages [10,17]. A smart contract has a state, a constructor that is called when a smart contract is first deployed in the blockchain, and a set of functions that could alter the state of the smart contract. The smart contract constructor gets executed once a smart contract is deployed resulting in instantiating a smart contract object in the blockchain. The constructor initializes the smart contract object and uses the implicit parameters sent alongside the smart contract deployment message to initialize the owner of the smart contract and the assets' value sent to this smart contract. Miners verify that the end-user who deploys a smart contract indeed owns these assets. Once assets are sent to a smart contract, the ownership of these assets is moved to the smart contract itself. Smart contract assets can only be transacted on within the smart contract logic until these assets are unlocked from the smart contract as a result of a smart contract function call. To execute a smart contract function, end-users submit their function call accompanied by the function parameters through messages to miners. These messages could include implicit parameters as well (e.g., *msg.sender*). Miners execute[2] the function on the current state of the contract and record any contract state changes in their current block in the blockchain. Therefore, a smart contract object state might span many blocks after the block where the smart contract is first deployed.

3 Atomic Cross-Chain Transaction Model

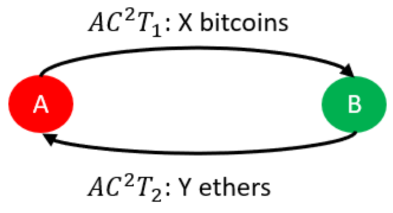

Figure 1: An atomic cross-chain transaction graph to swap X bitcoins for Y ethers between Alice (A) and Bob (B).

[1]Forks and fork resolutions are discussed in later sections.

[2]End-users pay to miners a smart contract deployment fee plus a function invocation fee for every function call.

An Atomic Cross-Chain Transaction, AC^{2}T, is a distributed transaction to transfer the ownership of assets stored in multiple blockchains among two or more participants. This distributed transaction consists of sub-transactions and each sub-transaction transfers an asset on some blockchain. An AC^{2}T is modeled using a directed graph $\mathcal{D} = (\mathcal{V}, \mathcal{E})$ [16] where $\mathcal{V}$ is the set of vertexes and $\mathcal{E}$ is the set of edges in $\mathcal{D}$. $\mathcal{V}$ represents the participants in AC^{2}T and $\mathcal{E}$ represents the sub-transactions in AC^{2}T. A directed edge $e = (u, v) \in \mathcal{E}$ represents a sub-transaction that transfers an asset $e.a$ from a source participant $u \in \mathcal{V}$ to a recipient participant $v \in \mathcal{V}$ in some blockchain $e.BC$. Figure 1 shows an example of an AC^{2}T graph between Alice (A) and Bob (B). As shown, the edge (A, B) represents the sub-transaction AC^2T_1 that transfers X bitcoins from A to B while the edge (B, A) represents the sub-transaction AC^2T_2 that transfers Y ethers from B to A.

An atomic cross-chain commitment protocol is required in order to correctly execute an AC^{2}T. This protocol must ensure the atomicity and the commitment of all sub-transactions in AC^{2}T as follows.

- Atomicity: either all or none of the asset transfers of all sub-transactions in the AC^{2}T take place.
- Commitment: once the atomic cross-chain commitment protocol decides the commitment of an AC^{2}T, asset transfers of all sub-transactions in this AC^{2}T must eventually take place.

An atomic cross-chain commitment protocol is a variation of the two phase commit protocol (2PC) [8, 14]. Therefore, we use the analogy of 2PC to explain an abstraction of the atomic cross-chain commitment protocol. In 2PC, a distributed transaction spans multiple data partitions and each partition is responsible for executing a sub-transaction. A coordinator sends a vote request to all involved data partitions. Upon receiving a vote request, a data partition votes back *yes* only if it succeeds in executing all the operations of its sub-transaction on the involved data objects. Otherwise, a data partition votes *no* to the coordinator. A coordinator decides to commit a distributed transaction if all involved data partitions vote yes, otherwise it decides to abort the distributed transaction. If a commit decision is reached, all data partitions commit their sub-transactions. However, if an abort is decided, all data partitions must abort their sub-transactions. 2PC assumes that the coordinator and the data partitions are trusted. The main challenge in blockchain systems is how to design a trust-free variation of 2PC where participants do not trust each other and a protocol cannot depend on a centralized trusted coordinator.

Consider an AC^{2}T $\mathcal{A}$. An atomic cross-chain commitment protocol requires that for every edge $e = (u, v) \in \mathcal{E}$, the source participant u locks an asset $e.a$ in Blockchain $e.BC$. This asset locking is necessary to temporarily prevent the participant u from spending $e.a$ through other transactions in $e.BC$. If every source participant u locks $e.a$ in $e.BC$, the atomic cross-chain commitment protocol can decide to commit $\mathcal{A}$. Once the protocol decides to commit $\mathcal{A}$, every recipient participant v should be able to *redeem* the asset $e.a$. However, if the protocol decides to abort $\mathcal{A}$ because some participants do not comply with the protocol or a participant requests the transaction to abort, every source participant u should be able to get a refund of their locked assets $e.a$.

In blockchain systems, smart contracts are used to implement this logic. For each edge $e = (u, v)$, participant u deploys a smart contract SC_e in Blockchain $e.BC$ to lock an asset $e.a$ owned by u in SC_e. SC_e ascertains to conditionally transfer $e.a$ to v if a commitment decision is reached, otherwise $e.a$ is refunded to u. A smart contract SC_e exists in one of three states: *published* (P), *redeemed* (RD), or *refunded* (RF). A smart contract SC_e is *published* if it gets deployed to $e.BC$ by u. Publishing the smart contract SC_e serves *two* important goals towards the atomic execution of $\mathcal{A}$. First, it represents a *yes* vote on the sub-transaction corresponding to the edge e. Second, it locks the asset $e.a$ in blockchain $e.BC$. A smart contract SC_e is *redeemed* if participant v successfully redeems the asset $e.a$ from SC_e. Finally, a smart contract SC_e is *refunded* if the asset $e.a$ is refunded to participant u.

Now, if for every edge $e = (u, v) \in \mathcal{E}$, the participant u publishes a smart contract SC_e in $e.BC$, it means that all participants vote yes on $\mathcal{A}$, lock their involved assets in $\mathcal{A}$, and hence $\mathcal{A}$ can commit. However, if some participants decline to publish their smart contracts, $\mathcal{A}$ has to abort. The commitment of $\mathcal{A}$ requires the redemption of **every** smart contract SC_e in $\mathcal{A}$. On the other hand, if $\mathcal{A}$ aborts, this requires the refund of **every** smart contract SC_e in $\mathcal{A}$.

To implement conditional smart contract redemption and refund, a cryptographic commitment scheme primitive based on [12] is used. A *commitment scheme* allows a user to commit to some chosen value without revealing this value. Once this hidden value is revealed, other users can verify that the revealed value is indeed the one that is used in the commitment. A *hashlock* is an example of a commitment scheme. A hashlock is a cryptographic one-way hash function $h = H(s)$ that is used to conditionally lock assets in a smart contract using h, the lock, until a hash secret s, the key, is revealed. Once s is revealed, everyone can verify that lock h equals $H(s)$ and hence unlocks the assets locked in the smart contract.

An atomic cross-chain commitment protocol should ensure that smart contracts in $\mathcal{A}$ are *either all redeemed or all refunded*. For this, a protocol uses *two* mutually exclusive commitment scheme instances: a redemption commitment scheme and a refund commitment scheme. All smart contracts in $\mathcal{A}$ commit their redemption action to the redemption commitment scheme instance and their refund action to the refund commitment scheme instance. If the protocol decides to commit $\mathcal{A}$, the protocol must publish the redemption commitment scheme secret. This allows all participants in $\mathcal{A}$ to redeem their assets. However, if the protocol reaches an abort decision, the protocol must publish the refund commitment scheme secret. This allows participants in $\mathcal{A}$ to refund the locked assets in every published smart contract. A protocol must ensure that once the secret of one commitment scheme instance is revealed, the secret of the other instance cannot be revealed. This guarantees the *atomic* execution of $\mathcal{A}$.

Algorithm 1 illustrates a smart contract template that can be used in implementing an atomic cross-chain commitment protocol. Each smart contract has a sender s and recipient r (Line 2), an asset a (Line 3) to be transferred from s to r through the contract, a state (Line 4), and a redemption and refund commitment scheme instances rd

and rf (Lines 5 and 6). A smart contract is published in a blockchain through a deployment message. When published, its constructor (Line 7) is executed to initialize the contract. The deployment message of a smart contract typically includes some implicit parameters like the sender's address (msg.sender, Line 8) and the asset value (msg.val, Line 9) to be locked in the contract. The constructor initializes the addresses, the asset value, the refund and redemption commitment schemes, and sets the contract state to P (Line 11).

Algorithm 1 An atomic swap smart contract template

```
abstract class AtomicSwapSC {
 1: enum State {Published (P), Redeemed (RD), Refunded
    (RF)}
 2: Address s, r // Sender and recipient public keys.
 3: Asset a
 4: State state
 5: CS rd // Redemption commitment scheme
 6: CS rf // Refund commitment scheme
 7: procedure CONSTRUCTOR(Address r, CS rd, CS rf)
 8:     this.s = msg.sender, this.r = r
 9:     this.a = msg.val
10:     this.rd = rd, this.rf = rf
11:     state = P
12: end procedure
13: procedure REDEEM(Evidence $evd_{rd}$)
14:     requires(state == P and IsRedeemable($evd_{rd}$))
15:     transfer a to r, state = RD
16: end procedure
17: procedure REFUND(Evidence $evd_{rf}$)
18:     requires(state == P and IsRefundable($evd_{rf}$))
19:     transfer a to s, state = RF
20: end procedure
21: procedure ISREDEEMABLE(Evidence $evd_{rd}$)
22:     return verify(rd, $evd_{rd}$)
23: end procedure
24: procedure ISREFUNDABLE(Evidence $evd_{rf}$)
25:     return verify(rf, $evd_{rf}$)
26: end procedure
}
```

In addition, each smart contract has a redeem function (Line 13) and a refund function (Line 17). A redeem function takes an evidence parameter. This evidence parameter proves that the decision is to commit AC^{2}T. The redeem function requires the smart contract to be in state P and that the provided evidence is a valid redemption commitment scheme secret (Line 14). If all these requirements hold, the asset a is transferred from the contract to the recipient and the contract state is changed to *RD*. However, if any requirement is violated, the redeem function fails and the smart contract state remains unchanged.

Similarly, the refund function requires the smart contract to be in state P and that the provided evidence is a valid refund commitment scheme secret (Line 18). If all these requirements hold, the asset a is refunded from the contract to the sender and the contract state is changed to *RF*.

The redeem and the refund functions use *two* helper functions: IsRedeemable (Line 21) and IsRefundable (Line 24). IsRedeemable verifies that the provided evidence is a valid redemption commitment scheme secret and hence the smart contract can be redeemed. Similarly, IsRefundable verifies that the provided evidence is a valid refund commitment scheme secret and hence the smart contract can be refunded.

4 AC^3: Atomic Cross-Chain Commitment

This section presents AC^{3}WN, an **A**tomic **C**ross-**C**hain **C**ommitment (AC^3) protocol that achieves both **atomicity** and **commitment** of an AC^{2}T. First, Section 4.1 presents an important building block on how miners of one blockchain validate the publishing of a transaction or a smart contract in another blockchain. Second, we present AC^{3}WN, an AC^3 protocol that uses a permissionless **W**itness **N**etwork to coordinate AC^{2}Ts in Section 4.2. Also, Section 4.2 explains how the witness network miners ensure the integrity of a smart contract code deployed in another blockchain. Using a permissionless network of witnesses does not require more trust in the witness network than the required trust in the blockchains used to exchange the assets in an AC^{2}T. Furthermore, the AC^{3}WN protocol overcomes the vulnerability of centralized solutions that are subject to failures and denial of service attacks.

4.1 Cross-Chain Validation

This section explains different techniques for how the miners of one blockchain, *the validators*, can validate the publication and verify the state of a smart contract deployed in another blockchain, *the validated* blockchain. A **simple but impractical** solution is to require all the miners of every blockchain to serve as validators of all other blockchains. A blockchain validator maintains a copy of the validated blockchain and for every newly mined block, a validator validates the mined block and adds it to its local copy of the validated blockchain. If all mining nodes mine one blockchain and validate all other blockchains, mining nodes can consult their local copies of these blockchains to validate the publishing and hence verify the state of any smart contract in any blockchain. If a participant needs the miners of the validator blockchain to validate the publishing of a smart contract in some other validated blockchain, this participant submits evidence that comprises the block id and the transaction id of the smart contract in the validated blockchain to the miners of the validator blockchain. This evidence is easily verified by the mining node of the validator blockchain by consulting their copy of the validated blockchain. However, this full replication of all the blockchains in all the mining nodes is impractical. Not only does it require massive processing power to validate all blockchains, but also it requires significant storage and network capabilities at each mining node.

Alternatively, miners of one blockchain, the validators, can run *light nodes* [9] of other blockchains, the validated blockchains. A light node, as defined in [9], is a node that downloads only block headers of the validated blockchain, verifies the proof of work of these block headers, and downloads only the blockchain branches that are associated with transactions of interest to this node. This solution requires the validators to mine for one blockchain and run light nodes for every validated blockchain. The validators can consult their local light node copy of the validated blockchain to validate the publishing and hence verify the state of a smart contract in the validated blockchain. Although the cost of maintaining a light node is much cheaper than maintaining a blockchain full copy, running a light node for all blockchains does not scale as the number of blockchains increases.

The previous two techniques put the onus of validating one blockchain on the miners of another blockchain. In addition, they require changes in the current infrastructure by requiring the miners of one blockchain to either maintain a full copy or a light node of other blockchains.

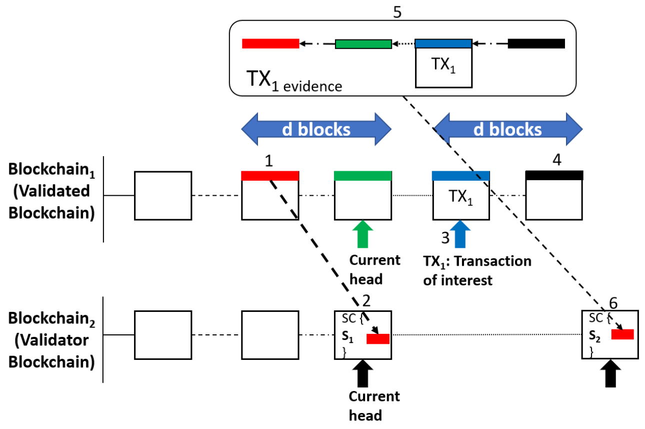
5
TX_1
TX_1 evidence
d blocks
d blocks
1
4
Blockchain$_1$ (Validated Blockchain)
TX_1
3
Current head
TX_1: Transaction of interest
2
6
Blockchain$_2$ (Validator Blockchain)
SC { S_1 }
SC { S_2 }
Current head

Figure 2: How miners of one blockchain could validate transactions in another blockchain.

Our proposal: Another way to allow miners of one blockchain, the validators, to validate the publication and verify the state of a smart contract in another blockchain, the validated, is to push the validation logic into the code of a smart contract in the validator blockchain. A smart contract in the validator blockchain is deployed and stores the header of a *stable block* in the validated blockchain. A stable block is a block at depth d from the current head of the validated blockchain such that the probability of forking the blockchain at this block is negligible (e.g., a block at depth ≥ 6 in the Bitcoin blockchain [2]). A participant who deploys the smart contract in the validator blockchain stores the block header of a stable block of the validated blockchain as an attribute in the smart contract object in the validator blockchain. When the transaction of the smart contract of interest takes place in a block in the validated blockchain and after this block becomes a stable block, at depth d, a participant can submit evidence of the transaction occurrence in the validated blockchain to the miners of the validator blockchain. This evidence comprises the headers of all the blocks that follow the stored stable block header in the smart contract of the validator blockchain in addition to the block where the transaction of interest took place. The evidence is submitted to the validator smart contract via a function call. This smart contract function validates that the passed headers follow the header of the stable block previously stored in the smart contract object and that the proof of work of each header is valid. In addition, the function verifies that the transaction of interest indeed took place and that the block of this transaction is stable and buried under d blocks in the validated blockchain.

Figure 2 shows an example of a validator blockchain, blockchain$_2$, that validates the occurrence of transaction TX_1 in the validated blockchain, blockchain$_1$. In this example, there exists a smart contract SC that gets deployed in the current head block of blockchain$_2$ (labeled by number 2 in Figure 2). SC has an initial state S_1 and stores the header of a stable block, at depth d, in blockchain$_1$ (labeled by number 1). This header is represented by a red rectangle inside SC. SC's state is altered from S_1 to S_2 if evidence is submitted to miners of blockchain$_2$ that proves that TX_1 took place in blockchain$_1$ in some block after the stored stable block header in SC. When TX_1 takes place in blockchain$_1$ (labeled by number 3) and its block becomes a stable block at depth $\geq d$ (labeled by number 4), a participant submits the evidence (labeled by number 5) to the miners of blockchain$_2$ through SC's function call (labeled by number 6). This function takes the evidence as a parameter and verifies that the submitted blocks took place after the stored stable block in SC. This verification ensures that the header of each submitted block includes the hash of the header of the previous block starting from the stored stable block header in SC. In addition, this function verifies the proof of work of each submitted block header. Finally, the function validates that TX_1 took place in some block in the submitted evidence and that this block has already become a stable block. If this verification succeeds, the state of SC is altered from S_1 to S_2. This technique allows miners of one blockchain to verify transactions and smart contracts in another blockchain without maintaining a copy of this blockchain. In addition, this technique puts the evidence validation responsibility on the developer of the validator smart contract.

4.2 AC^{3}WN: Permissionless Witness Network

This section presents AC^{3}WN, an AC^3 protocol that uses a *permissionless blockchain network* of witnesses to decide whether an AC^2T should be committed or aborted. **Miners** of this blockchain are collectively the **witnesses** on AC^2Ts. The main design challenge of the AC^{3}WN protocol is how to use a permissionless network of witnesses to implement the redemption and refund commitment scheme instances used by every smart contract in AC^{2}T. In addition, how to ensure that the two instances are *mutually exclusive*.

When a set of participants want to execute an AC^{2}T, they deploy a smart contract SC_w in the witness network where SC_w is used to coordinate the AC^{2}T. SC_w has a state that determines the state of the AC^{2}T. SC_w exists in one of three states: *Published* (P), *Redeem_Authorized* (RD_{auth}), or *Refund_Authorized* (RF_{auth}). Once SC_w is deployed, SC_w is initialized to the state P. If the witness network decides to commit the AC^{2}T, the witnesses set SC_w's state to RD_{auth}. However, if the witness network decides to abort the AC^{2}T, the witnesses set SC_w's state to RF_{auth}.

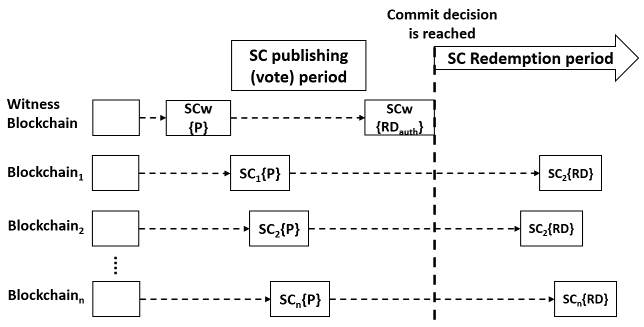

Figure 3: Coordinating AC^2Ts using a permissionless witness network.

Figure 3 shows an AC^{2}T that exchanges assets among blockchains, $blockchain_1, ..., blockchain_n$ and uses a *witness blockchain* for coordination. Also, it illustrates the AC^{3}WN protocol steps. For every AC^{2}T, a directed graph $\mathcal{D} = (\mathcal{V}, \mathcal{E})$ is constructed at some timestamp t and multisigned by all the participants in the set $\mathcal{V}$ generating a graph multisignature $ms(\mathcal{D})$ as shown in Equation 1. The timestamp t is important to distinguish between identical AC^2Ts among the same participants. The order of participant signatures in $ms(\mathcal{D})$ is not important. Any signature order indicates that all participants in the AC^{2}T agree on the graph $\mathcal{D}$ at some timestamp t.

$$ms(\mathcal{D}) = sig(..., sig(sig((\mathcal{D}, t), p_1), p_2), ..., p_{|\mathcal{V}|}) \qquad (1)$$

A participant registers $ms(\mathcal{D})$ in a smart contract SC_w in the witness network where SC_w's state is initialized to P. The state P indicates that participants of the AC^{2}T agreed on $\mathcal{D}$. In addition, participants agree to conditionally link the redeem and the refund actions of their smart contracts in the AC^{2}T to SC_w's states RD_{auth} and RF_{auth} respectively. Note that once SC_w gets deployed in the witness network, the SC_w object gets a unique address that can be used to reference SC_w in the redemption and the refund actions of smart contracts that transfer assets among participants. Afterwards, the participants *parallelly* deploy their smart contracts in the blockchains, $blockchain_1, ..., blockchain_n$, as shown in Figure 3. After all the participants deploy their smart contracts in the AC^{2}T, a participant may submit a state change request to the witness network miners to alter SC_w's state from P to RD_{auth}. This request is accompanied by evidence that all smart contracts in the AC^{2}T are deployed and correct. Upon receiving this request, witness network miners verify that SC_w's state is P and that participants of the AC^{2}T have indeed deployed their smart contracts in the AC^{2}T in their corresponding blockchains. In addition, miners verify that all these smart contracts are in state P and that the redemption and the refund of these smart contracts are conditioned on SC_w's states RD_{auth} and RF_{auth} respectively (via SC_w's unique address). Finally, witness network miners verify the integrity of each deployed smart contract in AC^{2}T through a multi-step process. Since each smart contract is a class and each smart contract deployment is an object instantiation of this class, witness network miners need to verify that the object attributes match their corresponding description in the graph $\mathcal{D}$ and the object functions have not been tampered with by the smart contract deployer (asset sender). First, witness network miners verify that the attributes of each smart contract such as sender, recipient, and asset amount match the attribute values of the edge corresponding to this smart contract $\forall e = (u, v) \in \mathcal{D}.\mathcal{E}$. Then, witness network miners must ensure that the recipient of each transfer smart contract has signed the smart contract functions and that this signature is included in one of the attributes of the smart contract object. This ensures that each recipient has verified the smart contract transfer functions and that the deployer (the sender) of each transfer smart contract cannot tamper with the agreed upon smart contract functions (e.g., redeem and refund functions).

If this verification succeeds for every smart contract in AC^{2}T, witness network miners record SC_w state change to RD_{auth} in their current block. Once a block that reflects the state change of SC_w to RD_{auth} is mined in the witness network, the commitment of the AC^{2}T is decided and participants can use this block as commitment evidence to redeem their assets in the smart contracts of the AC^{2}T. The commit decision is illustrated in Figure 3 using the vertical dotted line.

Similarly, if some participants decline to deploy their smart contracts in the AC^{2}T or a participant changes her mind before the commitment of the AC^{2}T, a participant can submit a state change request to the witness network miners to alter SC_w's state from P to RF_{auth}. The miners of the witness network only verify that SC_w's state is P. If this verification succeeds, the miners of the witness network record SC_w's state change to RF_{auth} in their current block. Once a block that reflects the state change of SC_w to RF_{auth} is mined in the witness network, the AC^{2}T is considered aborted and the participants can use this block as evidence of the abort to refund their assets in the deployed smart contracts of the AC^{2}T. Note that SC_w is programmed to ensure that SC_w's state can only be changed either from P to RD_{auth} or from P to RF_{auth} but no other state transition is allowed. This ensures that SC_w's states RD_{auth} and RF_{auth} are mutually exclusive. Miners use the cross-chain evidence validation techniques presented in Section 4.1 to validate the deployment of smart contracts in other blockchains.

Algorithm 2 presents the details of SC_w. SC_w consists of *four* functions: Constructor (Line 6), AuthorizeRedeem (Line 12), AuthorizeRefund (Line 16), and VerifyContracts (Line 20). The Constructor initializes SC_w with the participants' public keys, the multisigned graph of the AC^{2}T, and the headers of stable blocks in all the involved blockchains in AC^{2}T. This information is necessary for the witness network miners to later verify the correctness and the deployment of all smart contracts in the AC^{2}T. AuthorizeRedeem alters SC_w's state from P to RD_{auth}. To call AuthorizeRedeem, a participant provides evidence of the deployment of all the smart contracts in the AC^{2}T (Line 12). AuthorizeRedeem first verifies that SC_w's state is currently P. In addition, AuthorizeRedeem verifies that all smart contracts in the AC^{2}T are published and correct through a VerifyContracts function call (Line 13). If this verification succeeds, SC_w's state is altered to RD_{auth} (Line 14). On the other hand, AuthorizeRefund verifies only that the state of SC_w is P (Line 17). If true, SC_w's state is altered to RF_{auth} (Line 18).

VerifyContracts validates that all smart contracts in the AC^{2}T are published and correct. For every edge $e = (u, v) \in \mathcal{D}.\mathcal{E}$, VerifyContracts first verifies the evidence evd_e corresponding to the edge e (Line 25) as explained in Section 4.1. Then, VerifyContracts finds the matching smart contract SC_e in evd_e. Then, VerifyContracts ensures that SC_e matches its description in the edge e by checking that the smart contract sender is u, the recipient is v, the asset value is $e.a$, and the redeem and refund commitment schemes are set to SC_w's states RD_{auth} and RF_{auth} respectively (Lines 27 - 29). Also, VerifyContracts checks the integrity of SC_e's functions by checking that the code of SC_e's functions matches the signature of the recipient v stored in SC_e (Line 30). If any verification step fails, VerifyContracts fails and sets the return value to *false* (Line 32). However, if all smart contracts in the provided list are correct, VerifyContracts sets the return value to *true* (Line 21). VerifyContracts ensures that AuthorizeRedeem cannot be exe-

Algorithm 2 Witness network smart contract as an AC^{2}T Coordinator.

```
class WitnessSmartContract {
 1: enum State {Published (P), Redeem_Authorized ($RD_{auth}$), Refund_Authorized ($RF_{auth}$)}
 2: Address [] pk // Addresses of all participants in AC$^2$T
 3: Mutlisignature ms // The multisigned graph $\mathcal{D}$
 4: State state
 5: Header[] h // Stable block headers of blockchains in $AC^2T$
 6: procedure CONSTRUCTOR(Address[] pk, MS ms($\mathcal{D}$), Header[] h)
 7:     this.pk = pk
 8:     this.ms = ms($\mathcal{D}$)
 9:     this.state = P
10:     this.h = h
11: end procedure
12: procedure AUTHORIZEREDEEM(Evidence evd )
13:     requires (state == P and VerifyContracts(evd))
14:     this.state = $RD_{auth}$ // Commit Decision
15: end procedure
16: procedure AUTHORIZEREFUND
17:     requires (state == P)
18:     this.state = $RF_{auth}$ // Abort Decision
19: end procedure
20: procedure VERIFYCONTRACTS(Evidence evd)
21:     valid = true
22:     $\mathcal{D}$ = decrypt(ms, pk)
23:     for each $e = (u, v) \in \mathcal{D}.\mathcal{E}$ do
24:         let $evd_e \in evd$ be e's evidence
25:         Verify PoW and hash integrity of each block header in $evd_e$ to ensure they follow $h[e]$ // h[e] is the header of the stable block of e's blockchain e.BC.
26:         Let $SC_e \in evd_e$ be the smart contract of e
27:         Verify $SC_e.s == u$, $SC_e.r == v$, $SC_e.a == e.a$
28:         Verify $SC_e.rd$ links to this smart contract state
29:         Verify $SC_e.rf$ links to this smart contract state
30:         Verify v's signature on $SC_e$ functions
31:         if any verification fails then
32:             valid = false
33:         end if
34:     end for
35:     return valid
36: end procedure
}
```

cuted unless all smart contracts in the AC^{2}T are deployed and correct and hence a commit decision can be reached.

Algorithm 3 presents a smart contract class inherited from the smart contract template in Algorithm 1 in order to use SC_w's state as redemption and refund commitment scheme secrets. The commitment scheme is represented as a pair (SC_w, d) where SC_w is the address of the witness network coordination smart contract and d is required depth to ensure that SC_w and its state transition are stable. IsRedeemable returns *true* if SC_w's state is RD_{auth} (Line 8), while IsRefundable returns *true* if SC_w's state is RF_{auth} (Line 14). As the witness network is permissionless, forks could possibly happen resulting in *two* concurrent blocks where SC_w's state is RD_{auth} in the first branch and SC_w's state is RF_{auth} in the second branch. To avoid atomicity violations, participants cannot use a witness network block where SC_w's state is RD_{auth} or RF_{auth} in their smart contract redemption and refund respectively unless this block is buried under at least d blocks in the witness network. As the probability of a fork at depth d (e.g., 6 blocks in the Bitcoin network [2]) is negligible, SC_w's state eventually converges to either RD_{auth} or RF_{auth}.

Algorithm 3 Smart contract for permissionless AC^3.

```
class PermissionlessSC extends AtomicSwapSC {
 1: Signature $sig_r$ // Recipient integrity signature on Smart Contract function codes.
 2: procedure CONSTRUCTOR(Address r, SC $SC_w$, Depth d, Signature $sig_v$)
 3:     this.rd = this.rf = ($SC_w$, d)
 4:     this.$sig_r$ = $sig_v$
 5:     super(r, this.rd, this.rf) // parent constructor
 6: end procedure
 7: procedure ISREDEEMABLE(Evidence evd)
 8:     if evd is valid, the state of $SC_w \in evd$ is $RD_{auth}$, and $SC_w$'s state update block is at depth$\geq d$ then
 9:         return true
10:     end if
11:     return false
12: end procedure
13: procedure ISREFUNDABLE(Evidence evd)
14:     if evd is valid, the state of $SC_w \in evd$ is $RF_{auth}$, and $SC_w$'s state update block is at depth$\geq d$ then
15:         return true
16:     end if
17:     return false
18: end procedure
}
```

The following steps summarizes the AC^{3}WN protocol steps to execute the AC^{2}T shown in Figure 1:

1. Alice and Bob construct the AC^2T's graph $\mathcal{D}$ and multisign $(\mathcal{D}, t)$ to generate $ms(\mathcal{D})$.
2. Either Alice or Bob registers $ms(\mathcal{D})$ in a smart contract SC_w and publishes SC_w in the witness network setting SC_w's state to P. SC_w follows Algorithm 2.
3. Afterwards, Alice publishes a smart contract SC_1 using Algorithm 3 to the Bitcoin network that states the following:
    - Move X bitcoins from Alice to Bob if Bob provides evidence that SC_w's state is RD_{auth}.
    - Refund X bitcoins from SC_1 to Alice if Alice provides evidence that SC_w's state is RF_{auth}.
    - SC_1 contains Bob's signature on the code of SC_1's functions: Redeem, Refund, IsRedeemable, and IsRefundable.
4. Concurrently, Bob publishes a smart contract SC_2 to the Ethereum network using Algorithm 3 stating the following:
    - Move Y ethers from Bob to Alice if Alice provides evidence that SC_w's state is RD_{auth}.
    - Refund Y ethers from SC_2 to Bob if Bob provides evidence that SC_w's state is RF_{auth}.
    - SC_2 contains Alice's signature on the code of SC_2's functions: Redeem, Refund, IsRedeemable, and IsRefundable.

5. After both SC_1 and SC_2 are published, any participant can submit a state change request of SC_w from P to RD_{auth} to the witness network miners. This request is accompanied by evidence that SC_1 and SC_2 are published in the Bitcoin and the Ethereum blockchains respectively. The witness network miners first verify that SC_w's state is currently P. Then, they verify that both SC_1 and SC_2 are published and correct in their corresponding blockchains. If these verifications succeed, the miners of the witness network record SC_w's state change to RD_{auth} in their current block. Once a block that reflects the state change of SC_w to RD_{auth} is mined and gets buried under d blocks in the witness network, Alice and Bob can use this block as evidence to redeem their assets from SC_2 and SC_1 respectively.
6. If a participant declines to publish a smart contract, the other participant can submit a state change request of SC_w from P to RF_{auth} to the witness network miners. The witness network miners verify that SC_w's state is currently P. If true, miners record SC_w's state change to RF_{auth} in their current block. Once a block that reflects the state change of SC_w to RF_{auth} is mined and gets buried under d blocks in the witness network, Alice and Bob can use this block as evidence to refund their assets from SC_1 and SC_2 respectively.

This protocol uses two blockchain techniques to ensure that SC_w's states RD_{auth} and RF_{auth} are *mutually exclusive*. First, it uses the smart contract programmable logic to ensure that SC_w's state can only be altered from P to RD_{auth} or from P to RF_{auth}. Second, it uses the longest chain fork resolution technique to resolve forks in the witness network blockchain. This ensures that in the rare case of forking where one branch has SC_w's state of RD_{auth} and the other branch has SC_w's state of RF_{auth}, the fork is eventually resolved resulting in either SC_w's state is RD_{auth} or SC_w's state is RF_{auth} but not both.

5 AC³WN Analysis

This section analyzes the AC^{3}WN protocol introduced in Section 4.2. First, we establish that the proposed protocol ensures atomicity. Then we analyze the scalability of the witness network and how it affects the scalability of the commitment protocol. Finally, we explain how this protocol extends the functionality of previous proposals in [16, 23].

5.1 AC³WN: Atomicity Correctness Proof

The correctness of the AC^{3}WN protocol depends on the stability of the underlying blockchains. For example, the stability of the Bitcoin [22] network requires that at least a majority of the mining nodes be honest and adhere to the Bitcoin's mining protocol. Also, as explained in [22], the probability of forking the Bitcoin network at depth d blocks exponentially decreases as d increases assuming the majority of mining nodes are honest. Therefore, the correctness of the AC^{3}WN on the Bitcoin network requires the same assumptions needed for the Bitcoin network stability. First, we prove the correctness of the AC^{3}WN protocol assuming that the witness network and the asset networks do not fork in Lemma 5.1. Then, Lemma 5.2 proves that atomicity violations are negligible in the AC^{3}WN protocol if the probability of forking in the witness network and the asset networks at depth d is negligible.

Lemma 5.1. *Assume no forks in the witness network, then the AC^{3}WN protocol is atomic.*

Proof. Consider an AC^{2}T executed by the AC^{3}WN protocol and assume the atomicity of this transaction is violated. This atomicity violation implies that there exist two smart contracts SC_i and SC_j in AC^{2}T where SC_i is redeemed and SC_j is refunded. The redemption of SC_i implies that there exists a block in the witness network where SC_w's state is RD_{auth} while the refund of SC_j implies that there exists a block in the witness network where SC_w's state is RF_{auth}. Since SC_w is programmed to only allow state transitions either from P to RD_{auth} or from P to RF_{auth}, the two function calls to alter SC_w's state from P to RD_{auth} and from P to RF_{auth} cannot take effect in one block. Miners of the witness network shall accept one and reject the other. Therefore, these two state changes must be recorded in two separate blocks. As there exist no forks in the witness network, one of these two blocks must *happen before* the other. This implies that either SC_w's state is altered from RD_{auth} in one block to RF_{auth} in a following block or altered from RF_{auth} in one block to RD_{auth} in a following block. However, only state transitions from P to RD_{auth} or from P to RF_{auth} are allowed and no other state transition is permitted leading to a contradiction. $\square$

Lemma 5.2. *Let the probability of forks at depth d in the permissionless witness network be negligible $< \epsilon$, then AC^{3}WN protocol is atomic with a probability $> 1 - \epsilon$.*

Proof. Contrapositive: assume an AC^{2}T executed by the AC^{3}WN protocol and the atomicity of this transaction is violated with a probability $p > \epsilon$. This atomicity violation implies that there exists two smart contract SC_i and SC_j in AC^{2}T where SC_i is redeemed and SC_j is refunded. The redemption of SC_i implies that there exists a block in the witness network where SC_w's state is RD_{auth} while the refund of SC_j implies that there exists a block in the witness network where SC_w's state is RF_{auth}. As SC_w's states RD_{auth} and RF_{auth} are conflicting states, this implies that the block where SC_w's state update to RD_{auth} occurs must exist in a fork from the block where SC_w's state update to RF_{auth} occurs. The atomicity violation of the AC^{2}T with a probability p implies that the fork probability in the witness network must be $p > \epsilon$. Hence, AC^{3}WN is atomic with a probability $> 1 - \epsilon$ $\square$

5.2 The Scalability of AC³WN

One important aspect of AC^3 protocols is *scalability.* Does using a permissionless network of witnesses to coordinate AC^{2}Ts limit the scalability of the AC^{3}WN protocol? In this section, we argue that the answer is *no.* To explain this argument, we first develop an understanding of the properties of executing AC^{2}Ts and the role of the witness network in executing AC^{2}Ts.

An AC^{2}T is a distributed transaction that consists of sub-transactions. Each sub-transaction is executed in a blockchain. An AC^3 protocol coordinates the atomic execution of these sub-transactions across several blockchains. An AC^3 protocol must ensure the atomic execution of the distributed transaction. This atomic execution of a distributed transaction requires the ACID [13, 15] execution of every sub-transaction in this distributed transaction in addition to the atomic execution of the distributed transaction itself. The ACID execution of a sub-transaction executed within a single blockchain is guaranteed by the miners of this blockchain. Miners use many techniques including mining, verification, and the miner's rationale to join the longest chain in order to implement ACID executions of transactions within a single blockchain. The atomicity of a distributed transaction is the responsibility of the distributed transaction coordinator. Therefore, the main role of the witness network in the AC^{3}WN protocol is to ensure the atomicity of the AC^{2}T. Since the atomicity coordination of AC^{2}Ts is *embarrassingly parallel*, different witness networks can be used to coordinate different AC^{2}Ts.

Assume two concurrent AC^{2}Ts, t_1 and t_2. The atomic execution of t_1 does not require any coordination with the atomic execution of t_2. Each AC^{2}T requires its witness network to ensure that either all sub-transactions in the AC^{2}T are executed or none of them are. Therefore, t_1 and t_2 do not have to be coordinated by the same witness network. t_1 can be coordinated by one witness network while t_2 can be coordinated by another witness network. If t_1 and t_2 conflict at the sub-transaction level, this conflict is resolved by the miners of the blockchain where these sub-transactions are executed. Therefore, using a permissionless witness network to coordinate AC^{2}Ts does not limit the scalability of the AC^{3}WN protocol. Different permissionless networks can be used used to coordinate different AC^{2}Ts. For example, the Bitcoin network can be used to coordinate t_1 while the Ethereum network can be used to coordinate t_2.

5.3 Handling Complex AC^{2}T Graphs

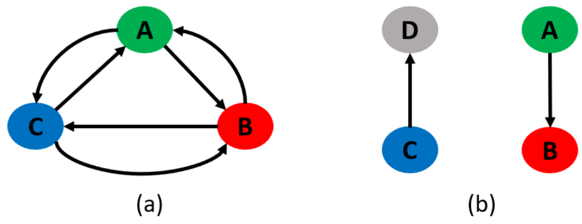

Figure 4: Examples of complex graphs handled by the AC^{3}WN protocol: (a) cyclic and (b) disconnected.

One main improvement of the AC^{3}WN protocol over the state-of-the-art AC^3 protocols in [16,23] is its ability to coordinate the atomic execution of AC^{2}Ts with complex graphs. This improvement is achieved because the AC^{3}WN protocol does not depend on the rational behavior of the participants in the AC^{2}T to ensure atomicity. Instead, the protocol depends on a permissionless network of witnesses to coordinate the atomic execution of AC^{2}Ts. Once the participants agree on the AC^{2}T graph and register it in the smart contract SC_w in the witness network, participants cannot violate atomicity as the commit and the abort decisions are decided by the state of SC_w. The state transitions of SC_w are witnessed and verified by the miners of the witness network. Therefore, the publication order of the smart contracts in the AC^{2}T cannot result in an advantage to any coalition among the participants. Participants can concurrently publish their smart contracts in the AC^{2}T, both in Figures 1 and 4, without worrying about the maliciousness of any participant.

Figure 4 illustrates two complex graph examples that either cannot be atomically executed by the protocols in [16, 23] or require additional mechanisms and protocol modifications to be atomically executed. These graphs appear in supply-chain applications. Both Nolan's and Herlihy's single leader protocol require the AC^{2}T graph to be acyclic once the leader node is removed. Therefore, both protocols fail to execute the transaction graph shown in Figure 4a. Removing any node from the graph in Figure 4a still results in a cyclic graph. Herlihy presents a multi-leader protocol in [16] to handle cyclic graphs. However, both Nolan's and Herlihy's protocols fail to handle disconnected graphs similar to the graph shown in Figure 4b. On the other hand, the AC^{3}WN protocol ensures the atomic execution of AC^{2}Ts irrespective of the AC^{2}T's graph structure.

6 Evaluation

This section analytically compares the performance and the overhead of the AC^{3}WN protocol to the state-of-the-art atomic swap protocol presented by Herilhy in [16]. First, we compare the latency of AC^{2}Ts as the diameter of the transaction graph $\mathcal{D}$ increases in Section 6.1. Then, the monetary cost overhead of using a permissionless network of witnesses to coordinate a AC^{2}T is analyzed in Section 6.2. Afterwards, an analysis on how to choose the witness network is developed in Section 6.3. Finally, an analysis of the AC^{2}T throughput as the witness network is chosen from the top-4 permissionless cryptocurrencies, sorted by market cap, is presented in Section 6.4.

6.1 Latency

The AC^{2}T latency is defined as the difference between the timestamp t_s when an AC^{2}T is started and the timestamp t_c when the AC^{2}T is completed. t_s marks the moment when participants in the AC^{2}T start to agree on the AC^{2}T graph $\mathcal{D}$. t_c marks the completion of all the asset transfers in the AC^{2}T by redeeming all the smart contracts in AC^{2}T.

Let Δ be enough time for any participant to publish a smart contract in any permissionless blockchain, or to change a smart contract state through a function call of this smart contract, and for this change to be publicly recognized [16]. Also, let $Diam(\mathcal{D})$ be the AC^{2}T graph diameter. $Diam(\mathcal{D})$ is the length of the longest path from any vertex in $\mathcal{D}$ to any other vertex in $\mathcal{D}$ including itself.

The single leader atomic swap protocol presented in [16] has *two* phases: the AC^{2}T smart contract sequential deployment phase and the AC^{2}T smart contract sequential redemption phase. The deployment phase requires the deployment of all smart contracts in the AC^{2}T, N, where exactly $Diam(\mathcal{D}) \leq N$ smart contracts are sequentially deployed resulting in a latency of $\Delta \cdot Diam(\mathcal{D})$. Similarly, the redemption phase requires the redemption of all smart contracts in the AC^{2}T, N, where exactly $Diam(\mathcal{D}) \leq N$ smart contracts are sequentially redeemed resulting in a latency of $\Delta \cdot Diam(\mathcal{D})$. The overall latency of an AC^{2}T that uses this protocol equals the latency summation of these two phases $2 \cdot \Delta \cdot Diam(\mathcal{D})$. Figure 5 visualizes the two phases of the protocol where time advances from left to right. As shown, some smart contracts (e.g., SC_2, SC_3, and SC_4) could be deployed and redeemed in parallel but there are exactly $Diam(\mathcal{D})$ sequentially deployed and $Diam(\mathcal{D})$ sequentially redeemed smart contracts resulting in an overall latency of $2 \cdot \Delta \cdot Diam(\mathcal{D})$. Note that the protocol allows the parallel deployment and redemption of some smart contracts as long as they do not lead to an advantage to either a participant or a coalition of participants in the AC^{2}T.

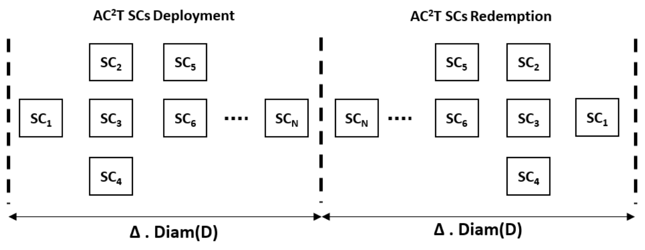

Figure 5: The overall transaction latency of $2 \cdot \Delta \cdot Diam(\mathcal{D})$ when the single leader atomic swap protocol in [16] is used.

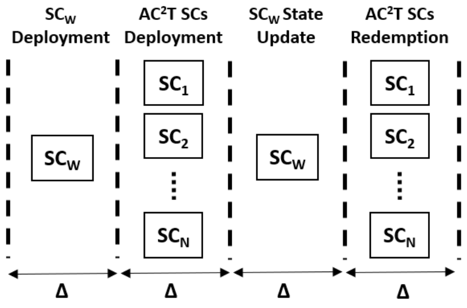

Figure 6: The overall transaction latency of $4 \cdot \Delta$ when the AC^{3}WN protocol in Section 4.2 is used.

On the other hand, the AC^{3}WN protocol has *four* phases: the witness network smart contract deployment phase, the AC^{2}T smart contract parallel deployment phase, the witness network smart contract state change phase, and the AC^{2}T smart contract parallel redemption phase. The witness network smart contract deployment requires the deployment of the smart contract SC_w in the witness network resulting in a latency of Δ. The AC^{2}T smart contract parallel deployment requires the parallel deployment of all smart contracts, N, in the AC^{2}T resulting in a latency of Δ. The witness network smart contract state change requires a state change in SC_w either from P to RD_{auth} or from P to RF_{auth} through SC_w's AuthorizeRedeem or AuthorizeRefund function calls resulting in a latency of Δ. Finally, the AC^{2}T smart contract parallel redemption requires the parallel redemption of all smart contracts, N, in the AC^{2}T resulting in a latency of Δ. The overall latency of an AC^{2}T that uses this protocol equals to the latency summation of these four phases $4 \cdot \Delta$. Figure 6 visualizes the four phases of the protocol where time advances from left to right. As shown, all smart contracts in the AC^{2}T are parallelly deployed and parallelly redeemed resulting in an overall latency of $4 \cdot \Delta$.

Figure 7 compares the overall AC^{2}T latency in Δs resulting from Herlihy's protocol in [16] and our protocol in Section 4.2 as the transaction graph diameter, $Diam(\mathcal{D})$ increases. As shown, our protocol achieves a constant latency of $4 \cdot \Delta$ irrespective of the transaction diagram value while Herlihy's protocol achieves a linearly increasing latency as the transaction diameter value increases. Note that the smallest transaction graph consists of *two* nodes and *two* edges and hence the graph diameter in Figure 7 starts at 2.

6.2 Cost Overhead

This section analyzes the monetary cost overhead of the AC^{3}WN protocol in comparison to Herlihy's atomic swap protocol in [16]. As explained in Section 2, miners charge end-users a fee for every smart contract deployment and

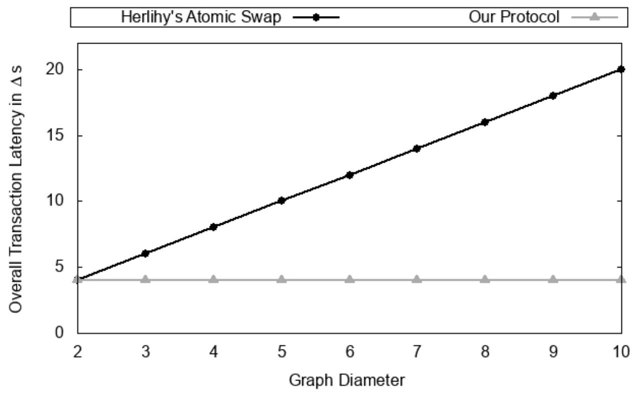

Figure 7: The overall AC^{2}T latency in Δs as the graph diameter, $Diam(\mathcal{D})$, increases.

every smart contract function call that results in a smart contract state change. This fee is necessary to **incentivize** miners to add smart contracts and append smart contract state changes to their mined blocks. As shown in Figures 5 and 6, both protocols deploy a smart contract for every edge $e \in \mathcal{D}.\mathcal{E}$. This results in the deployment of $N = |\mathcal{E}|$ smart contracts in the smart contract deployment phase of both protocols. In addition, both protocols invoke a redemption or a refund function call for every deployed smart contract in the AC^{2}T resulting in N function calls. However, the AC^{3}WN protocol requires the deployment of an additional smart contract SC_w in the witness network in addition to an additional function call to change SC_w's state either from P to RD_{auth} or from P to RF_{auth}. The cost of the deployment and state transition function call of SC_w comprises the monetary cost overhead of the AC^{3}WN protocol. Let f_d be the deployment fee of any smart contract $SC_i \in AC^2T$ and f_{fc} be the function call fee of any smart contract function call. Then, the overall AC^{2}T fee of Herlihy's protocol is $N \cdot (f_d + f_{fc})$ while the overall AC^{2}T fee of the AC^{3}WN protocol is $(N + 1) \cdot (f_d + f_{fc})$. This analysis shows that AC^{3}WN imposes a monetary cost overhead of $\frac{1}{N}$ the transaction fee of Herilhy's protocol assuming equal deployment and functional call fees for all the smart contracts in the AC^{2}T.

But, *how much does it cost in dollars to deploy a smart contract and make a smart contract function call?* The answer is, it depends. Many factors affect a smart contract fee such as the length of the smart contract and the average transaction fee in the smart contract's blockchain [6,27]. Ryan [27] shows that the cost of deploying a smart contract with a similar logic to SC_w's logic in the Ethereum network costs approximately \$4 when the ether to USD rate is \$300. Currently, this costs approximately \$2 assuming the current ether to USD rate of \$140.

6.3 Choosing the Witness Network

This section develops some insights into choosing the witness network for an AC^{2}T. This choice has to consider the risk of choosing different permissionless blockchain networks as the witness of an AC^{2}T and the relationship between this risk and the value of the assets exchanged in this AC^{2}T. As the state of the witness smart contract SC_w determines the state of an AC^{2}T, forks in the witness network present a risk to the atomicity of the AC^{2}T. A fork in the witness network where one block has SC_w's state of RD_{auth} and another block has SC_w's state of RF_{auth} might result in an atomicity violation leading to asset losses of some participants in the AC^{2}T. To overcome possible violations, our AC^{3}WN protocol does not consider a block where SC_w's state is either RD_{auth} or RF_{auth} as a commit or an abort evidence until this block is buried under d blocks in the witness network. This technique of resolving forks by waiting is presented in [22] and used by Pass and Shi in [25] to eliminate uncertainty of recently mined blocks. This fork resolution technique is efficient as the probability of eliminating a fork within d blocks is sufficiently high. Waiting for d confirmations (blocks) is an essential technique to confirm the commitment of any transaction in any open blockchain that could fork. **Therefore, waiting for d block confirmations is not a replacement for Herihly's and Nolan's timelocks.** In fact, the timelocks of Nolan's and Herlihy's should consider waiting for d confirmations after every smart contract deployment. Recall the atomic swap example in Section 1. Alice deploys a smart contract SC_1 in the Bitcoin network to transfer X bitcoins to Bob. Immediately afterwards, Bob deploys SC_2 in the Ethereum network to transfer Y ethers to Alice. Now, what happens if Alice redeems the Y ethers from SC_2 and the Bitcoin network gets forked before the deployment of SC_1 (or the other way around). Even if timelocks are respected, this fork could result in an atomicity violation. Now, the question is whether waiting for d confirmations is sufficient to make Herlihy's protocol [16] atomic? The answer is no. Even if the smart contract deployments are buried under d blocks, the violation of timelocks due to crash failures or network denial of service at Bob could result in a refund of the bitcoins in SC_1 resulting in an atomicity violation. Herlihy's protocol assumes synchrony of the network and atomicity could be violated even if there are no forks in the network.

A malicious participant in an AC^{2}T could fork the witness blockchain for d blocks in order to steal the assets of other participants in the AC^{2}T. To execute this attack, a malicious participant rents computing resources to execute a 51% attack on the witness network. The cost of an hour of 51% attack for different cryptocurrency blockchains is presented in [7]. If the cost of running this attack for d blocks is less than the expected gains from running the attack, a malicious participant is incentivized to act maliciously.

To prevent possible maliciousness, the cost of running a 51% attack on the witness network for d blocks must exceed the potential gains of running the attack. Let V_a be the value of the potentially stolen assets if the attack succeeds. Also, let C_h be the hourly cost of a 51% attack on the witness network. Finally, let d_h be the expected number of mined blocks per hour for the witness blockchain (e.g., $d_h = 6$ blocks / hour for Bitcoin). The value d must be set to ensure that V_a is less than the cost of running the attack for d blocks $\frac{d \cdot C_h}{d_h}$. Therefore d must ensure the inequality $d > \frac{V_a \cdot d_h}{C_h}$ in order to disincentivize maliciousness. For example, let V_a be \$1M and assume Bitcoin is used for coordination. The cost per hour of a 51% attack on Bitcoin is approximately $C_h = \$300K$. Therefore, d must be $> \frac{\$1M \cdot 6}{\$300K} = 20$.

6.4 Throughput

The throughput of the AC^{2}Ts is the number of transactions per second (tps) that could be processed assuming that every AC^{2}T spans a fixed set of blockchains and is witnessed by a fixed witness blockchain. For an AC^{2}T that spans multiple blockchains, the throughput is bounded by the slowest involved blockchain in the AC^{2}T including the witness network. Let tps_i be the throughput of blockchain i. The throughput of the AC^{2}Ts that span blockchains i, j, ..., n and are witnessed by the blockchain w equals $min(tps_i, tps_j .., tps_n, tps_w)$. Table 1 shows the throughput

Table 1: The throughput in tps of the top-4 permissionless cryptocurrencies sorted by their market cap.

Blockchain	*tps*	*Blockchain*	*tps*
1) Bitcoin	7	3) Litecoin	56
2) Ethereum	25	4) Bitcoin Cash	61

of the top-4 permissionless cryptocurrencies sorted by their market cap [24]. An example AC^{2}T that exchanges assets between Ethereum and Litecoin blockchains and is witnessed by Bitcoin achieves a throughput of 7. The witness network should be chosen from the set of involved blockchains (Litecoin and Ethereum in this example) to avoid limiting transaction throughput.

7 Conclusion

This paper presents AC^{3}WN, the first decentralized **A**tomic **C**ross-**C**hain **C**ommitment protocol that ensures the all-or-nothing atomicity semantics even in the presence of participant crash failures and network denial of service attacks. Unlike in [16,23] where the protocol correctness mainly relies on participants' rational behaviour, AC^{3}WN separates the coordination of an Atomic Cross-Chain Transaction, AC^{2}T, from its execution. A permissionless open network of witnesses coordinates the AC^{2}T while participants in the AC^{2}T execute sub-transactions in the AC^{2}T. This separation allows AC^{3}WN to ensure atomicity of all the sub-transactions in an AC^{2}T even in the presence of failures. In addition, this separation enables AC^{3}WN to parallelly execute sub-transactions in the AC^{2}T reducing the latency of an AC^{2}T from O($Diam(\mathcal{D})$) in [16], where $Diam(\mathcal{D})$ is the diameter of the AC^{2}T graph $\mathcal{D}$, to $O(1)$ irrespective of the size of the AC^{2}T graph $\mathcal{D}$. Also, this separation allows AC^{3}WN to scale by using different permissionless witness networks to coordinate different AC^{2}Ts. This ensures that using a permissionless network of witnesses for coordination does not introduce any performance bottlenecks. Finally, the AC^{3}WN protocol extends the functionality of the protocol in [16] by supporting AC^{2}Ts with complex graphs (e.g., cyclic and disconnected graphs). AC^{3}WN introduces a slight monetary cost overhead to the participants in the AC^{2}T. This cost equals the cost of deploying a coordination smart contract in the witness network plus the cost of a function call to the coordination smart contract to decide whether to commit or to abort the AC^{2}T. The smart contract deployment and function call approximately cost \$2 combined per AC^{2}T when Ethereum is used to coordinate the AC^{2}T.

8 Acknowledgement

This work is partially funded by NSF grants CNS-1703560 and CNS-1815733.

9 References

[1] Atomic cross-chain trading. `https://en.bitcoin.it/wiki/Atomic_cross-chain_trading`, 2018.

[2] Bitcoin confirmations. `https://www.buybitcoinworldwide.com/confirmations/`, 2018.

[3] Coinbase. `https://coinbase.com`, 2018.

[4] Robinhood. `https://robinhood.com/`, 2018.

[5] Solidity — solidity 0.5.5 documentation. `https://solidity.readthedocs.io/en/v0.5.5/`, 2018.

[6] Avg. transaction fee historical chart. `https://bitinfocharts.com/comparison/transactionfees-btc-eth.html`, 2019.

[7] Cost of a 51% attack for different cryptocurrencies. `https://www.crypto51.app/`, 2019.

[8] P. A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency control and recovery in database systems. 1987.

[9] V. Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

[10] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen. Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 303–312. ACM, 2017.

[11] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *NSDI*, pages 45–59, 2016.

[12] O. Goldreich. *Foundations of cryptography: volume 1, basic tools.* Cambridge University Press, Cambridge, 2001.

[13] J. Gray et al. The transaction concept: Virtues and limitations. In *VLDB*, volume 81, pages 144–154. Citeseer, 1981.

[14] J. N. Gray. Notes on data base operating systems. In *Operating Systems*, pages 393–481. Springer, 1978.

[15] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM computing surveys (CSUR)*, 15(4):287–317, 1983.

[16] M. Herlihy. Atomic cross-chain swaps. In *Proceedings of the 2018 ACM symposium on principles of distributed computing*, pages 245–254, 2018.

[17] M. Herlihy. Blockchains from a distributed computing perspective. *Communications of the ACM*, 62(2):78–85, 2019.

[18] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 279–296, 2016.

[19] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 17–30. ACM, 2016.

[20] S. Maiyya, V. Zakhary, D. Agrawal, and A. E. Abbadi. Database and distributed computing fundamentals for scalable, fault-tolerant, and consistent maintenance of blockchains. *PVLDB*, 11(12):2098–2101, 2018.

[21] S. Micali. Algorand: the efficient and democratic ledger. *CoRR, abs/1607.01341*, 2016.

[22] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[23] T. Nolan. Alt chains and atomic transfers. `https://bitcointalk.org/index.php?topic=193281.msg2224949#msg2224949`, 2013.

[24] D. O'Keeffe. Understanding cryptocurrency transaction speeds. `https://medium.com/coinmonks/understanding-cryptocurrency-transaction-speeds-f9731fd93cb3`, 2018.

[25] R. Pass and E. Shi. Hybrid consensus: Efficient consensus in the permissionless model. In *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[26] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[27] D. Ryan. Costs of a real world ethereum contract. `https://hackernoon.com/costs-of-a-real-world-ethereum-contract-2033511b3214`, 2017.

[28] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.