A Secure Consensus Protocol for Sidechains

Fangyu Gai, Cesar Grajales, Jianyu Niu and Chen Feng
School of Engineering
University of British Columbia (Okanagan Campus), Kelowna, Canada
{fangyu.gai, cesar.grajales, jianyu.niu and chen.feng}@ubc.ca

***Abstract*—Sidechain technology has been envisioned as a promising solution to accelerate today's public blockchains in terms of scalability and interoperability. By relying on the mainchain for security, different sidechains can formulate their own rules to reach consensus. Although the literature has considered the possibility of using consensus protocols in the sidechain, so far a tailor-made consensus protocol for sidechains with high performance and formal security proof has not been attempted. To fill this gap, we introduce *Vulcan*, a low overhead, highly efficient, security provable sidechain protocol. Vulcan makes use of smart contracts to ensure that only one block proposed in the sidechain will be enforced on the mainchain in each round, achieving consensus in an efficient manner. We give formal specifications of Vulcan which ensures safety and liveness with f validators (total number of $n \geq 2f + 1$) without online requirement of clients. For security analysis, we give formal security definitions and proofs under *Universally Composable Security* (UCS) model. As a proof of concept, we implement Vulcan and evaluate it in Ethereum testnet.**

I. Introduction

Recent years have witnessed a growing academic interest in the field of blockchain technology, which is the cornerstone behind cryptocurrencies such as Bitcoin [28] and Ethereum [35]. This technology has been envisioned as a trustless and decentralized platform on which developers are building various applications beyond cryptocurrencies. However, today's public blockchains (often referred to as permissionless blockchains, such as Bitcoin and Ethereum) are hindered by many challenges for web-scale adoption. One of the most fundamental challenges is scalability, which is caused by blockchain's nature of decentralization. In a permissionless blockchain, every node validates every transaction, which consequently imposes a fundamental limit on the throughput (i.e., the number of transactions that can be processed per second) [12]. In addition, since every single move of the execution of a transaction consumes resources (e.g., storage, bandwidth, computation, etc.) of the network, the fees charged by miners could be expensive.

A common approach to scale permissionless blockchains is to enact transactions "off-chain", i.e. execute transactions outside the blockchain. Off-chain solutions consider the existing blockchains as the base layer (referred to as "layer 1"), or the "mainchain", and arbitrary transaction functions as the upper layer (referred to as "layer 2"). In order to transfer from layer 1 to layer 2, participants are required to deposit some funds on-chain so that the money will be frozen on the mainchain and be "recreated" off-chain. After that, they update their off-chain status by performing arbitrary transactions following the off-chain protocols. Malicious behaviors could happen in off-chain networks, but when this situation happens, participants could complain to the mainchain by providing justifications which are verified by the mainchain. Therefore, the mainchain plays the role of a "judge" by running smart contracts which bridge the mainchain and the upper protocol. By minimizing the use of the underlying blockchain itself, this off-chain approach can achieve "scale-out" throughput as the throughput will increase as the size of the network grows [32].

A. Motivation

One off-chain proposal is called payment channel networks (PCN) [31] which enable two users to perform unlimited transactions off-chain only touching the mainchain during the deposit and withdrawal process. In a nutshell, two parties open a payment channel by making some deposits to the mainchain and then they can do off-chain transactions without touching the mainchain until they want to close the channel. Furthermore, even if two users have no direct payment channel established, they can still pay each other through a set of intermediaries. While numerous contributions have been made to improve the performance of PCN [27][13][21], they still face multiple challenges such as costly routing, expensive channel setup and rebalance requirement. Due to these obstacles, PCNs are mainly used in peer-to-peer micropayment scenarios.

In this paper, we focus on a different approach called sidechains [2], which has much wider application scenarios because it allows any participant to build a sidechain with specific purpose. Sidechains also share a deposit-withdraw scheme like PCN, but instead of building a channel between two parties, a sidechain is a new blockchain with its own rules and two-way pegged[1] with the mainchain. Another difference to PCN is that in order to guarantee the correctness of the sidechain, block proposers make commitments to the mainchain periodically, enforcing the state of the sidechain in the mainchain [18]. To summarize, a sidechain protocol is essentially a compression mechanism that periodically executes transactions, persists state, and anchors it in the mainchain, thereby achieving scale-out throughput.

Plasma [30] and NOCUST [22] are two of the most promising sidechain approaches which employ the smart contract technology to achieve two-way peg. They allow a centralized operator to manage the sidechain (or "commit-chain" in NOCUST) while guaranteeing that users always retain control over their assets. On one hand, they can achieve the same magnitude of throughput as centralized fiat payment processing systems, such as MasterCard and Visa. On the other hand, they ensure that end-users can always exit the sidechain if the operator is compromised as long as the mainchain is secure. Despite the

[1] Two-way peg is a cross-chain mechanism where assets are transferred between two blockchains at a fixed or deterministic exchange rate [2]

above two clear advantages, sidechain-based solutions are still in its infancy, facing several limitations.

- *A failure-prone operator.* Although the operator can be trustless, its failure or misbehavior still has critical impact on user experience. In particular, once the operator is compromised, so is the sidechain. Users have to perform mass-exit in order to keep their assets safe, which will probably cause congestion in the mainchain.
- *Withholding attack.* This attack, originated from *Selfish Mining*, has emerged as a critical attack against PoW-based blockchains [3] [19] [29], where selfish miners withhold blocks for unfair mining competition. In the sidechain context, the withholding attack consists of the block proposer forging an arbitrary block, not propagating it to other nodes, but making the commitment to the mainchain. From the mainchain's perspective, the committed block is always accepted since it does not verify the correctness of its content. Current solutions include either instructing the users to exit the sidechain or forcing the proposer providing the data through the smart contract. Both two solutions require users to be online and the latter will involve multiple on-chain transactions.
- *Online requirement.* Existing off-chain proposals, including sidechains and PCN, require the users to be online to receive the payment or to verify the block at the end of each block time for trustless operation. Otherwise, sidechain assets might be stolen. This requirement is impractical to fulfill. There are some potential solutions such as "watch towers" [26], where users could outsource their data to a trusted third party. However, this introduces additional trust assumptions and costs.

B. Our Solutions and Contributions

In order to address the above issues, we propose to design new consensus algorithms for sidechains. Although some existing work [12][30] has discussed the possibility of consensus mechanisms for sidechain (e.g., PoS, PoA, PBFT, etc.), a consensus protocol specially designed for sidechains with formal security proof is still missing in the literature. To fill this research gap, we present Vulcan: a low overhead, high efficiency, security provable sidechain protocol. To the best of our knowledge, Vulcan is the first formal specification of a sidechain-based protocol with high performance and rigorous security proof. More specifically, Vulcan[2] (i) allows organizations to run separate sidechains and provides various off-chain services to their end-users, and (ii) gives a guarantee to clients that their assets on sidechains are always safe without online requirement and they can always exit with their coins back to the mainchain. The key idea behind Vulcan is that we can leverage mainchain security properties in the design of sidechain consensus, leading to high-performance protocols which are previously impossible. Our contributions can be summarized as follows:

A BFT-based consensus algorithm specially designed for sidechain. We developed a new algorithm to solve the well-known Byzantine consensus problem [23] in the sidechain context, where the creation of a block is stricted to a fixed set of n authority nodes, called *validators* in our protocol, among which a maximum of $f < \frac{n}{2}$ can be Byzantine. To avoid inconsistency, each committed block is enforced on the mainchain through smart contracts and considered as a *checkpoint*. A checkpoint is only considered valid when it is approved by more than half validators. Thus we adopt the *aggregate signature* to assemble multiple signatures, i.e., approvals, from different validators into a single signature. We also employ a "lazy-challenge" mechanism to delay the verification of the checkpoint until some party complains. Since each pending checkpoint can be viewed on the mainchain, every party can run the verification locally and request a challenge when they find the checkpoint is invalid. A validator will be removed out of the committee if it is successfully challenged. Therefore, in order to stay in the committee, a block proposer has to reveal the content of the block to other validators for approval, which prevents withholding attack.

Safe Cross-Chain Transfer. Apart from consensus algorithms, Vulcan supports safe cross-chain transfer. Users can easily issue a forward transfer (from mainchain to sidechain) to join the sidechain since every party can verify on-chain transactions. However, it becomes complicated vice versa because of the opacity of off-chain transactions. In Vulcan, the balances of each user are constructed in a Merkle Patricia Trie (MPT) with the address as the *key* and the balance as the *value*. Over a block time, the balances are enforced by the latest checkpoint, which includes the root hash of the MPT. Our protocol ensures that as long as the security assumption ($n \geq 2f + 1$) holds, users can always withdrawal/exit from the latest checkpoint by submitting the correct merkle path to the smart contract without online requirement.

Security Model and Formal Proof. To be able to rigorously analyze the security properties of our protocol, we employ a general-purpose definitional framework called the Universally Composable Security (UCS) framework by Canetti [10]. By using the UCS framework, we formally define the security properties of our protocol and provide detailed proof.

C. Outline

Section II presents the system model. Section III describes the protocol in a high level. Section IV gives a formal definition of Vulcan under UCS model. Section V evaluates the performance. Section VI discusses the related work and Section VII concludes this paper.

II. System Model

In this section we introduce basic notations, key elements, communication model, adversary model and assumptions used in our protocol.

A. Main Elements

Vulcan involves the following roles:

Mainchain: We introduce a tamper-proof and smart contract enabled ledger (e.g., Ethereum) as our mainchain denoted by $\mathbb{MC}$. In our protocol, $\mathbb{MC}$ serves as the base layer, which keeps all the data including smart contracts, accounts, balances and transactions.

[2] Vulcan is a fictional planet of the *Star Trek* universe. Vulcans, residents of that planet, are noted for their attempt to live by logic and reason with as little interference from emotion as possible.

Smart Contract: A smart contract is a piece of code deployed on $\mathbb{MC}$ acting as the bridge between $\mathbb{MC}$ and the sidechain (denoted by $\mathbb{SC}$) and providing dispute-solving services. We use $\mathbb{C}$ to denote a set of contracts designated to a specific $\mathbb{SC}$. In order to meet specific requirements of different sidechains, each sidechain has a set of exclusive smart contracts containing a group of functions.

Validators: A set of validators $\mathcal{V} = \{V_1, V_2, ..., V_n\}$ run validations and consensus algorithms in $\mathbb{SC}$, where n is the number of validators. Each of them is identified by a unique pair of public/private key in the sidechain network. Validators take turns to propose blocks, one block at an epoch. During each epoch, the validator who is responsible to propose the block is called the *leader*, while the rest are called *followers*. The order of rotation is prefixed.

Clients: A set of clients $\mathcal{C} = \{C_1, C_2, ..., C_m\}$ are actual users of $\mathbb{SC}$, where m is the number of clients inside $\mathbb{SC}$ network. They can share the same public/private key pair between $\mathbb{MC}$ and $\mathbb{SC}$. Clients issue transactions directly to validators. In order to increase the success rate of acceptance, clients can multicast transactions to a group of validators they trust.

B. Communication Model

We assume our protocol under a synchronous communication network, where there is a known upper bound on the message transmission delay. We assume all parties have access to $\mathbb{MC}$ and $\mathbb{C}$, which are always available. All the transactions sent to $\mathbb{MC}$ and $\mathbb{C}$ will be finalized within time Δ. We assume parties can instantaneously acquire the status of $\mathbb{MC}$ so that they are always aware of the current epoch of the protocol.

C. Adversary Model

We consider the adversary can compromise all the clients and f validators, where $f \leq \frac{n-1}{2}$. Faulty parties may deviate from their normal behavior in arbitrary ways, e.g., hardware/software crash, nonsensical read and write to $\mathbb{MC}$, colluding with each other to cheat honest parties, etc. However, we assume they do not have enough computation power and money to compromise $\mathbb{MC}$. We also assume that all messages are signed by their individual senders and adversaries can not break collision resistant hashes and signatures used both in $\mathbb{MC}$ and $\mathbb{SC}$.

D. Cryptografic Primitives

Our protocol utilizes *aggregate signatures* which enables distributed signing among n participants $\{P_1, P_2, ..., P_n\}$. An aggregate signature scheme consists of four algorithms: KeyGen, Sign, Combine, and Verify. The former two algorithms are the same as the standard signature scheme. Algorithm Combine takes a vector of n triples (pk_i, m_i, σ_i) as input, where pk_i is the public-key for C_i, m_i is the message, and σ_i is the signature signed on m_i. The output is a single aggregate signature QA, whose length is the same as a signature on a single message. Algorithm Verify takes n pairs (pk_i, m_i) and the aggregate signature QA as input and outputs "True" only if σ was generated as an aggregate of n valid signatures.

III. System Overview

In this section, we present Vulcan from a high-level overview. Analogous to Ethereum [35], Vulcan is also an account-based protocol, where each participant has an account referring to their properties (e.g., balance, non-fungible tokens, etc.). Therefore, the whole protocol can also be viewed as a state machine. For simplicity, in this paper, the state only refers to account balances. The protocol begins with a genesis state and proceeds by executing transactions periodically until it terminates.

The sidechain$\mathbb{SC}$ is set up by $\mathcal{V}$ deploying $\mathbb{C}$ on $\mathbb{MC}$, and any validator can be the owner. The owner of $\mathbb{C}$ sets an endpoint in $\mathbb{C}$ indicating when $\mathbb{SC}$ will be closed. When it passes the endpoint, the protocol will halt, after which users can exit with the balance according to the latest agreed checkpoint.

In Vulcan, we rely on a fixed set of validators to maintain $\mathbb{SC}$. After $\mathbb{C}$ is deployed on $\mathbb{MC}$, $\mathbb{C}$ will keep the public key as the id of each validator for future verification. Validators collect signed transactions from clients, execute them, and package them along with the outcome into a block during each epoch. The leader of current epoch should propose that block by broadcasting it to followers. If the leader gets majority of votes after validators running the mPoA consensus algorithm (cf. Sec. III-B), then he commits the digest of the block header, along with a *quorum certificate* (QC), which is an aggregate of approvals form followers, to $\mathbb{MC}$ via an online transaction to enforce the state of the epoch. A QC is valid only if it is an aggregate of more than f signatures on the same hash of the block header from different validators (including the leader itself). Meanwhile, the validators publish the whole content of that block for every parties to query through an API.

A Byzantine leader might propose different blocks at a time and send them to different followers. In this case, the leader cannot cause a fork for two reasons: one leader can only commit once during its tenure and only one block can gain majority votes since non-faulty validators will not vote for two different blocks at a time. A validator ignores the inconsistency in the committed block and the received block as long as the former gains a valid QC. Another case is that a Byzantine leader might commit a faulty block without revealing it to the followers, performing withholding attack. To achieve this attack, the leader has to obtain an approval from at least one honest follower (assuming other f validators are compromised) to generate a valid QC. However, this cannot happen because an honest validator will not sign an incorrect block.

As for clients, they can always withdraw certain amount of money or exit with their assets from the latest checkpoint. Fig. 1 shows a system overview of Vulcan.

A. Epochs

Vulcan proceeds in epochs. An epoch is a slot of time during which a block is generated from transaction collection to the acceptance by $\mathbb{C}$. Epochs are numbered consecutively. Let e denote as the epoch number and l as the index of the leader of the corresponding epoch such that $l = e \mod |n|$. Note that a view-change (a new leader to collect transactions and propose it to followers for a new epoch) happens no matter

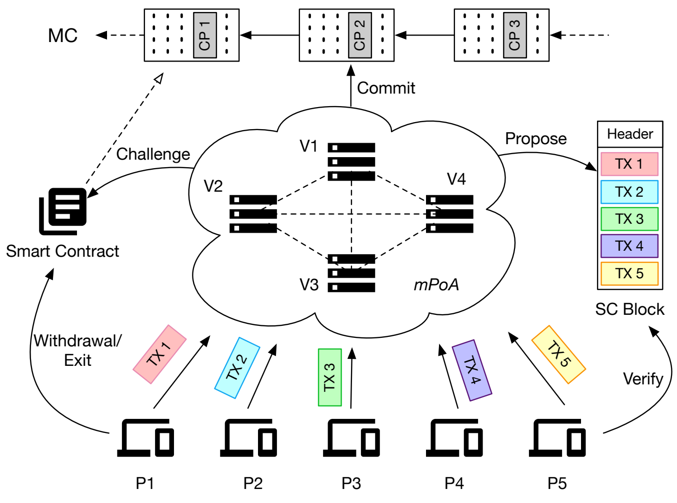

Fig. 1. An overview of transaction flow in Vulcan.

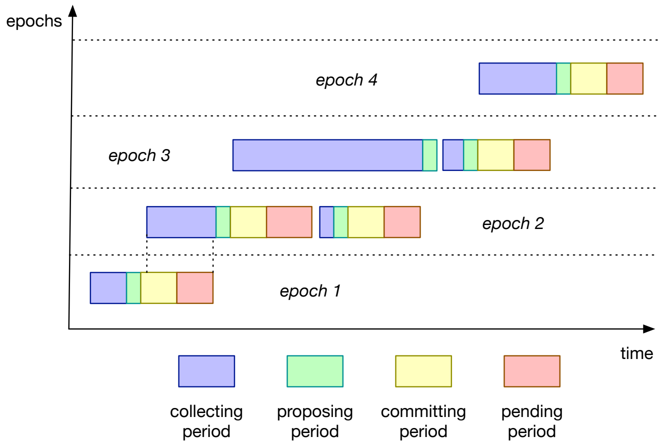

Fig. 2. An example of how epochs proceed

the epoch succeeds or not (more details are described in Sec. III-B). Each epoch has four terms described as follows:

- *Collecting term.* Validators collect transactions from clients during this term. This term starts at the beginning of the *committing term* and ends right after the success of last epoch.
- *Proposing term.* At the beginning of this term, the leader l packages the collected transactions from the last term into a block and proposes it by broadcasting the block to the followers. If the proposal fails (the leader receives less than f approvals within a constant time), this round is abandoned, and a new round (from collecting term) will begin. Otherwise, the epoch proceeds to the next term.
- *Committing term.* This term begins by the leader committing the block to $\mathbb{C}$. After the commitment being confirmed by $\mathbb{MC}$, the epoch proceeds to the pending term.
- *Pending term.* During this term, every participant from $\mathcal{V}$ and $\mathcal{C}$ can send *complaints* against the pending commitment, and it is the only term during which $\mathbb{C}$ handles *withdrawal/exit* requests from clients. If no messages are received during this term, then the epoch ends and the pending checkpoint is finalized. Otherwise, the duration of this term will be increased until the dispute is resolved (after a certain number of on-chain confirmations). If the challenge is successful, the current epoch will restart. Otherwise the protocol proceeds to epoch $e + 1$.

Note that two epochs overlap each other to reduce latency without causing any conflicts. An epoch starts after the end of the proposing term of the previous one. On the other hand, the proposing term of the next epoch starts after the previous epoch ends.

An example of how each epoch proceeds is depicted in Fig. 2. *Epoch* 1 shows a normal epoch, which ends in one round, no malicious events being observed. *Epoch* 2 starts at the beginning of the committing term of *epoch 1*, and the proposing term of *epoch* 2 starts after *epoch 1* ends. We can observe that the pending term of *epoch* 2 lasts longer than that of *epoch* 1. This is because the leader might be compromised and honest followers successfully "challenge" him. Hence, following that is a new round of *epoch* 2. Due to the long duration of *epoch* 2, the collecting term of *epoch* 3 is sustaining until the *epoch* 2 ends. However, the first round of *epoch* 3 fails at the proposing term, so another round comes instead. Then *epoch* 4 starts at the beginning of the committing term of *epoch 3* and survives 4 terms in one round.

We can learn from the above example that if every party is well-behaved, each epoch will only take one round to finish. Otherwise, it might take several rounds to finish. To ensure liveness, honest validators will vote out the leader if the epoch can not finish within certain rounds. We discourage malicious behaviors by instructing validators to deposit some money as collateral. If the leader is successfully challenged by more than f validators, it will be removed from the committee with monetary punishment. The design of punishment mechanism is the scope of this paper.

B. BFT-based Consensus

Vulcan adopts a BFT-based algorithm for validators to reach consensus. Other than tolerate up to $\frac{n-1}{3}$ Byzantine nodes, Vulcan can ensure security and liveness with a maximum of $\frac{n-1}{2}$ Byzantine nodes. Even though all of the validators are compromised (the protocol might halt unexpectedly), Vulcan still guarantees that funds in $\mathbb{SC}$ are safe under conditions that clients come online at least once per epoch to verify their balance (cf. Sec. VIII-A).

In Vulcan, we follow a *leader rotation* procedure which has been widely used to fairly distribute the responsibility of block production among validators [20][1]. The order of validators taking the lead is predetermined in $\mathbb{C}$ before deployed. We argue that it is nonsignificant how this order is decided. For example, we could follow the order of validators joining $\mathbb{SC}$. Once $\mathbb{C}$ is deployed, the order is settled unless some validators are voted out for Byzantine behaviors. The owner of $\mathbb{C}$ also determines a *validity time* indicating when $\mathbb{SC}$ is closed. When it passes the validity time, the protocol will halt waiting for clients to exit from the latest checkpoint. Then the funds will be fairly distributed and $\mathbb{SC}$ will terminate. The algorithm proceeds in epochs which are identified in Fig. 2. Every party in the sidechain has a global view of who is the leader of the current epoch by querying $\mathbb{C}$, and the leader will be automatically passed to the next one at the end of each epoch.

Algorithm 1 The consensus algorithm at v_i

```
Global State:
    $\mathcal{V} = \{v_1, v_2, ..., v_n\}$: the set of validators identified by their public keys
    current_leader: the public key of current leader

Local State:
    $Q_{txn} = \{tx_1, tx_2, ..., tx_k\}$: the set of transactions in the transaction queue, where $k$ is the total number

function PROPOSE():
    if $v_i$ =current_leader then
        $block$ :=execute($Q_{txn}$)
        $block$ :=sign($block$)
        broadcast($block$)
        $approvals$ :=receive()
        $QC$ :=combine($approvals$)
        if $approvals.length \geq \frac{n+1}{2}$ then
            commit($QC$)
        end if
    end if
end function

function DELIVER($block$):
    if verify($block$) then
        $approval$ :=approve($block$)
        $approval$ :=sign($approval$)
    end if
    send($approval$, current_leader)
end function
```

Algorithm 1 shows a simplified version of the consensus algorithm. We omit details of the verification of transactions and blocks (e.g., signature, timestamp, balance). Each validator maintains a transaction queue Q_{txn} locally. During the *collecting term* of each epoch, once a validator receives a transaction, he broadcasts it with a timestamp (indicating which epoch it belongs to) to the other validators after verifying its validity[3]. When the *collecting term* ends, the leader executes all transactions in Q_{txn} and constructs a block with the outcome (cf. Sec. III-C). Then he proposes that block by broadcasting it to all the followers (see function propose). On receiving the proposed block, followers verify its validity (see function deliver). If that block is correct, the validator approves this block by sending the leader an *approval* message which is the signature on the hash of the block header. If the leader receives more than f *approvals* from different validators (including itself), it aggregates the signatures into a single *aggregate signature* and commits it along with the digest of the block to $\mathbb{C}$ to create a checkpoint.

Each checkpoint consists of a hash of the committed block header, *aggregate signature*, and a *bit index*. We use *aggregate signature* as a proof that the committed block is approved by more than half of the validators. The *bit index* is an integer, every bit of which represents whether or not the indexed validator signed the *approval*. For example, we have 5 validators in total, indexed by v_0, v_1, v_2, v_3, v_4. The *bit index* of a certain checkpoint is 11 whose binary format is 01011, meaning that v_1, v_3, v_4 signed the *approvals*. Let $cp_i = \{H(B_i), QA(H(B_i)), \text{INDEX}\}$ denote the ith checkpoint, where $H(B_i)$ is the hash of the header of block B_i, $QA(H(B_i))$ is a constant-sized aggregate signature on $H(B_i)$, and INDEX is an integer which contains the indexes of a list of validators that signed the $QA(H(B_i))$. The basic workflow is as follows:

Aggregation. The leader calls the Combine function to aggregate m signatures on block B_i from m different validators into one. This signature will convince any verifier that the exact m validators approved the commitment of the block. There are several signature constructions to achieve signature aggregation [24][5][6]. In order to reduce the length of a checkpoint, we apply the one based on the short signature scheme of Boneh, Lynn, and Shacham [7].

Verification. We invoke the Verify function to execute the aggregate verification algorithm, verifying whether the committed block is actually approved by more than half of the validators. Given a checkpoint $cp_i = \{H(B_i), QA(H(B_i)), \text{INDEX}\}$ which can be queried from $\mathbb{C}$, verifiers first find a set of public keys via INDEX, then they verify that $QA(H(B_i))$ is a valid aggregate signature on $H(B_i)$ under the given keys.

We apply a *lazy-challenge* mechanism to verify the correctness of each checkpoint. After $\mathbb{C}$ receives the commitment, $\mathbb{C}$ will not verify the checkpoint until some party submits a *challenge*. Given that all parties can view the content of the pending checkpoint by querying $\mathbb{C}$, they will run the verification algorithm locally. Any party can submit a challenge before the *pending term* ends. On receiving a challenge, $\mathbb{C}$ will run the verification algorithm itself to make a binary decision. Either the leader or the challenger will win, and the loser will be punished (i.e., loose the bounty), but the details of punishment mechanism is not the focus of this paper.

In order to ensure the liveness of the protocol, validators can also submit *challenges* to vote out a non-responsive leader. If there is no block proposed or the proposed block is not committed after a local time-out, a validator can gossip the challenge message (i.e., essentially a signature) to other validators, which can be seen as a vote. Anyone who collects more than f votes aggregates them into one signature and submits the signature to $\mathbb{C}$ and let it decide how to deal with the leader.

C. Block Construction

A block in our protocol contains all the transactions happened in $\mathbb{SC}$ during a specific epoch[4], as well as the updated states (i.e., balances) after executing those transactions. As such, a block is a basic unit of a state transition in $\mathbb{SC}$. More precisely, after a block is published, either all of the transactions complete (all transactions are computed correctly), or they fail, and the state of $\mathbb{SC}$ remains unchanged.

[3] Malicious validators may hide transactions or ignore all the transactions from some particular participants. As such, participants could send their transactions to multiple validators or choose the one they trust the most.

[4] Transactions may be assigned to the next epoch due to network latency and Byzantine validators may arbitrarily discard transactions.

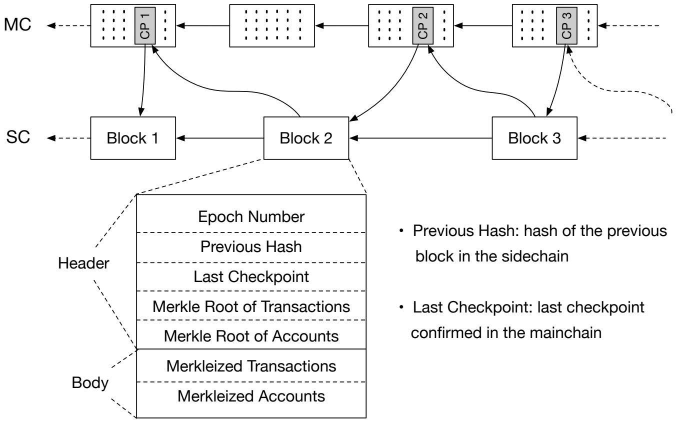

Fig. 3. The construction of $\mathbb{SC}$ blocks. $\mathbb{SC}$ is "stitched" onto $\mathbb{MC}$.

The design goal of constructing a block in $\mathbb{SC}$ is (i) to include all the information needed for a participant to withdraw or exit, (ii) that when a participant wants to withdraw or exit, $\mathbb{C}$ should be able to efficiently verify the proof of possession of the funds, and (iii) that all of the finalized blocks should follow a strict order according to when they are confirmed in $\mathbb{MC}$. To fulfill those requirements, Vulcan's blocks consist of a list of transactions, a list of updated accounts, along with the metadata required for verification. Fig. 3 depicts how $\mathbb{SC}$ blocks are constructed and "stitched" onto $\mathbb{MC}$.

1) Block Header: The block header contains metadata including the epoch number, a hash of the previous block, the reference of the last checkpoint confirmed on $\mathbb{MC}$, and the merkle roots of transactions and balances of accounts. The *last checkpoint* refers to the hash of the previous block's header. The reason to include this field in the block header is to provide a reference to the last checkpoint, and indicate this block is published after the previous block is confirmed on $\mathbb{MC}$.

2) Block Body: The block body consists of two merkle trees, a transaction tree and an account tree. The transaction tree is a regular *Merkle Tree* (MT) used in Bitcoin [28], where hashes of transactions forms leaf nodes and every non-leaf node is labelled with the hash of its two child nodes. Verifiers can efficiently and securely verify the existence of a leaf node in a large data structure by providing a MT proof which is the path consisting of all the hashes going up from the leaf node to the root. For the account tree, we introduce the *Merkle Patricia tree* (MPT) structure – used in Ethereum [35], to merkleize the accounts. In the context, the account id is the key and the balance of the account is the value in the MPT. Verifiers can efficiently and securely verify the existence of a key and the correctness of the expected value to the key using a MPT proof.

D. Withdrawal/Exit

The withdrawal/exit procedure can be viewed as a cross-chain protocol to transfer assets from $\mathbb{SC}$ to $\mathbb{MC}$. This part is more complex than deposit (i.e., transferring assets reversely) because the updates in $\mathbb{SC}$ are unknown to $\mathbb{MC}$. Therefore, participants have to provide sufficient proofs to $\mathbb{MC}$ about the funds they want to withdraw.

The difference between a withdrawal and an exit is that a participant withdraws a certain amount of money equal or less than the balance without notifying any malicious behaviors (from a participant's view), while a participant exit with all of his money left in $\mathbb{SC}$ when some obvious maliciousness happens. In Vulcan, participants withdraw or exit by sending a withdrawal/exit request to $\mathbb{C}$ containing the value to withdraw and a *Proof of Possession* (PoP) which is to prove a participant owns certain balance of assets in $\mathbb{SC}$. A standard PoP consists of following components:

- *Block Header.* It is a regular block header, whose hash should equal to the checkpoint.
- *Balance.* The balance of the account at the time of the checkpoint.
- *Merkle Path.* It is for $\mathbb{C}$ to verify whether the provided balance matches the expected value in the account tree with the MPT root included in the block header field.

Now, we start to describe the withdrawal/exit procedure assuming every party is honest. Parties can only withdraw from the current epoch. For example, after the *proposing term* of current epoch, the latest balance of C_i is x and it wants to withdraw y coins. C_i initiates the withdrawal by sending a withdrawal request directly to $\mathbb{C}$ containing a withdrawal value (i.e., y coins), as well as the PoP certifying he is the owner of x coins. After the verification, $\mathbb{C}$ unfreezes y coins on $\mathbb{MC}$ and send them to C_i. On another site, validators are always aware of the withdrawal. After it is finalized, validators will change the balance of C_i on $\mathbb{SC}$ to $x - y$ and generate a transaction indicating the balance change which will be included in the next block. An exit procedure works the same except indicating the withdrawal value.

If the number of Byzantine validators remains less than $\frac{n}{2}$, we assume that honest validators or clients can address malicious behaviors by issuing a *challenge* request (cf. Sec. III-B), even though the duration of the echo might be longer than usual. Therefore, participants can always withdraw/exit through a normal procedure. However, when the number of Byzantine validators is more than $\frac{n}{2}$ (e.g., all of the validators are compromised), it means it is no longer safe to stay in this $\mathbb{SC}$. At this point, end-users can go through an interactive procedure to exit, but we assume the users would verify their balances at the end of every epoch (cf. Appendix VIII-A).

IV. Formal Security Definition of Vulcan

A. Modeling Vulcan

In this section, we formally present our sidechain in the Universally Composable Security (UCS) framework. In particular, we follow the work of [17] applying a synchronous version of the global UC framework (GUC) [11], an extension of standard UC model allowing for a globally available set-up. Under this model, all the deposits (from $\mathbb{MC}$ to $\mathbb{SC}$) and the withdrawal/exit (from $\mathbb{SC}$ to $\mathbb{MC}$) are handled via a global ideal functionality $\mathcal{L}(\Delta)$, the state of which is globally accessible by all parties. $\mathcal{L}(\Delta)$ can freely add and remove money in user's accounts and the parameter Δ models that any interaction with $\mathcal{L}(\Delta)$ has a maximal delay Δ. A full definition of $\mathcal{L}(\Delta)$ can be found in [16].

In the UCS framework, a protocol is defined as a set of computing entities, each representing an actual component of the system, including an adversary (which models all the possible attacks that the protocol may suffer) and an environment (which models the external interactions that the protocol may have). The kind of computing entities that are used are *interactive Turing machines* (ITMs), which are basically traditional Turing machines with the added capability of interchanging information with other machines. Thus, the system of the present work is modeled as a protocol in the framework.

For a protocol to fulfill its intended security goals, two different protocols have to be designed: one representing an idealized model and one representing a real implementation. The ideal protocol will encompass all the functionality as a single entity, operating as desired, either normally or under any kind of attack. In the real protocol, each element is modeled as a separate computing entity. The security goal is achieved if any environment in which the protocol is deployed, cannot distinguish if it has interacted with the ideal model, or the realistic implementation.

The concept of distinction between the two protocols has a concrete definition, in terms of ensembles of probability distributions. An ensemble of probability distributions is a set of random variables $\mathcal{X} = \{X(z,k) : z \in \{0,1\}^*\}$, i.e., for all possible choices of $z \in \{0,1\}^*$ and $k \in \mathbb{N}$, there is a random variable $X(z,k)$ in the set.

Definition 1. We say that two probability distribution ensembles $\mathcal{X}$ and $\mathcal{Y}$ are *indistinguishable*, denoted by $\mathcal{X} \approx \mathcal{Y}$, if for any $a, b \in \mathbb{N}$, there exists $k_0 \in \mathbb{N}$ such that, for any $k > k_0$,

$$|P(X(z) = 1) - P(Y(z) = 1)| < \frac{1}{k^a} \quad (1)$$

with $z \in \{0,1\}^\lambda$ and $\lambda \leq k^b$.

Although this definition is slightly technical, in the context of the UCS framework, it essentially means that any polynomially bounded machine (PPT) in k (i.e., an ITM that is computationally bounded by a polynomial of k), cannot obtain different outputs after performing computations that are represented by random variables from the ensembles $\mathcal{X}$ and $\mathcal{Y}$ (for a given k and z), except with negligible probability, for an input z whose length is also polynomial in k.

We denote the ideal protocol with $\mathcal{I}$, and the real protocol with Π. The random variable that represents the output of the protocol Π in conjunction with an adversary $\mathcal{A}$ and an environment $\mathcal{E}$, with inputs k and z, is denoted by $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}}(k,z)$. To avoid complicating the notation, $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}}$ will represent the ensemble of probability distributions, for all possible choices of k and z. With this, we can state our security definition.

Definition 2. The concept of security of the protocol Π is achieved if for any adversary $\mathcal{A}$ and any environment $\mathcal{E}$, we can always find an adversary $\mathcal{S}$ (typically referred to as the "simulator") such that:

$$\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} \approx \text{EXEC}_{\mathcal{I},\mathcal{S},\mathcal{E}} \quad (2)$$

The ensembles of the execution of the protocol with environment $\mathcal{E}$ and adversary $\mathcal{A}$ are indistinguishable from the ensembles of the execution of the ideal functionality $\mathcal{I}$ with the same environment $\mathcal{E}$ and adversary $\mathcal{S}$. Part of the security proof, is to build a simulator $\mathcal{S}$ for any given adversary $\mathcal{A}$. Whenever the conditions of definition 2 are met, we say that Π UC-realizes the ideal $\mathcal{I}$.

Theorem 3. *There exists a protocol* Π*, such that it UC-realizes the ideal protocol* $\mathcal{I}$ *for the ideal funcionality* $\mathcal{F}$*.*

We assume synchronous communication network, where there is a known upper bound on the message transmission delay. To simplify the modeling, we assume that the operation of the protocol proceeds in units. The communication between any two of four computation entities takes exactly one unit (e.g., It takes one unit of time for a message sent from some party to reach $\mathbb{C}$ or other parties), while the communication between entities and the environment $\mathcal{E}$ takes zero units. As we describe the elements of the protocol, we present all the parameters that they require to work, and the mathematical notation for them.

B. Security and Efficiency Properties

We emphasize that our scheme provides both *safety* and *liveness* assuming no more than $\frac{n-1}{2}$ validators are faulty without online requirement. Even if the number of Byzantine validators is more than $\frac{n}{2}$, our scheme ensures safety by sacrificing liveness (i.e., the protocol might terminate) and it requires clients to be online at least once within an epoch (some techniques like watchtowers [26] can be used). Here, safety means that all honest parties can always retrieve the assets from $\mathbb{SC}$ to $\mathbb{MC}$. More specifically, safety can be divided in to the following security guarantees:

Consensus on sidechain update. A proposed block containing transactions happened during a single epoch should go through a consensus process, where more than a half of followers should approve that block before it is committed by the leader. If the committed block does not receive sufficient approvals, it might be reversed or the whole sidechain will be terminated.

Guaranteed withdrawal/exit. The end-users of a sidechain are guaranteed that they can always withdraw a certain amount of coins less than or equal to the balance, or exit the sidechain with all the assets from the latest agreed checkpoint.

In addition to safety, we introduce efficient goals by identifying the duration of an epoch in both optimistic and pessimistic cases. Every transaction[5] sent to $\mathbb{MC}$ is handled in time $O(\Delta)$ and the consensus time, denoted by τ (much smaller than Δ), is constantly set according to the use case. We can see from Fig. 2 that in optimistic case, one epoch can end in one round, which includes four distinct terms, while in pessimistic case, it might take several rounds (with limit to 3) to finish an epoch. Therefore, an epoch ends in time $O(\Delta + \tau)$ in both optimistic and pessimistic cases.

C. The Ideal Functionality

[5] Mainly depends on the average block production time of the MC, e.g., about 10 min in Bitcoin and 3 min in Ethereum.

Ideal Functionality $\mathcal{F}$

The ideal functionality $\mathcal{F}$ encompass all the functions of $V_1, \ldots, V_n$, $C_1, \ldots, C_m$, $\mathbb{C}$ and $\mathcal{L}$. It communicates with $\mathcal{E}$ via dummy parties $V_1^*, \ldots, V_n^*$, $C_1^*, \ldots, C_m^*$. For a fixed time Δ, $\mathcal{E}$ gives turns to the clients to make deposits (or join), make withdraws, exit or challenge the checkpoint. It also gives turns to the validators to also challenge the checkpoint.

(I) **Deposit**

A. Upon receiving $(\texttt{depositRequest}, \text{id}_{C_i^*}, x)$ from $\mathcal{E}$:

1) If the timer Δ has not expired, $\text{id}_{C_m^*} \notin \mathcal{C}^*$, and the mainchain balance for that ID is enough, invoke a dummy machine C_m^* for the client, update $\mathcal{C}^* := \mathcal{C}^* \cup \{\text{id}_{C_m^*}\}$, update the balance for the client $B(\text{id}_{C_i^*}, e) := x$ and send $(\texttt{endOfTurn})$ to $\mathcal{E}$.
2) If the timer Δ has not expired and $\text{id}_{C_i} \in \mathcal{C}$, send $(\texttt{existingClient})$ to $\mathcal{E}$.

B. Upon receiving $(\texttt{turnToTalk})$ for a deposit of an amount x through $\mathcal{C}_i^*$:

1) If the timer Δ has not expired, update the balance for the client $B(\text{id}_{C_i^*}, e) := B(\text{id}_{C_i^*}, e) + x$ and send $(\texttt{endOfTurn})$ to $\mathcal{E}$ through C_i^*.

(II) **Withdraw/Exit**

A. Upon receiving $(\texttt{turnToTalk})$ for a withdraw of an amount x from $\mathcal{E}$ through C_i:

1) If the timer Δ has not expired, verify if $B(\text{id}_{C_i^*}, e) \geq x$. If so, transfer x to C_i^* mainchain account, then send $(\texttt{endOfTurn})$ to $\mathcal{E}$. Else send $(\texttt{endOfTurn})$ to $\mathcal{E}$.

B. Upon receiving $(\texttt{turnToTalk})$ from $\mathcal{E}$ through C_i^* for it to exit:

1) If the timer Δ has not expired, verify if $B(\text{id}_{C_i^*}, e) \geq x$. If so, transfer x to C_i^* mainchain account and halt the machine C_i^*. Else, send $(\texttt{endOfTurn})$ to $\mathcal{E}$.
2) If $\mathcal{C}^* = \emptyset$, halt the machines $V_1^*, \ldots V_n^*$, send $(\texttt{executionEnd}, 1)$ if the remaining balance $b = 0$ or $(\texttt{executionEnd}, 0)$ if the remaining balance $b \neq 0$ and halt. Else, send $(\texttt{endOfTurn})$ to $\mathcal{E}$.

(III) **Transaction**

A. Upon receiving $(\texttt{turnToTalk})$ from $\mathcal{E}$ through any C_i^* and when timer Δ has expired:

1) Increase the epoch count $e := e + 1$, reset the transactions accumulators $b_i^s := 0$ and $b_j^r := 0$, the transactions queue $Q_{txn} = \emptyset$, reset the transactions count $n_{txn} := 0$, reset the transactions timer t_{txn} and send $(\texttt{Ready})$ to $\mathcal{E}$ to indicate that the Transactions term has started.

Upon receiving $(\texttt{turnToTransfer})$ from $\mathcal{E}$ through C_i to transfer an amount x to C_j^*:

1) Verify if $B_l(\text{id}_{C_i^*}, e) \geq x$. If so, update the accumulators $b_i^s := b_i^s + x$, $b_j^r := b_j^r + x$, the transactions queue $Q_{txn} := Q_{txn} \cup \{txn\}$, the counter $t_{txn} := t_{txn} + 1$. Repeat this n times to store an internal registry for each validator.
2) Evaluate if $t_{txn} \geq t_{max}$ or $n_{txt} \geq n_{max}$. If so, compare the j-th registry against the rest. If more than $\frac{n}{2}$ registries agree with the j-th, compile the block B_e and the checkpoint for the epoch with the information of the j-th registry, and send $(\texttt{blockReceived})$ to $\mathcal{E}$. If not, send $(\texttt{ready})$ to $\mathcal{E}$.

(IV) **Pending**

A. Upon receiving $(\texttt{blockReceived})$, $\mathcal{E}$ starts giving sequentially the turn to the validators and clients to request operations. Upon receiving $(\texttt{turnToTalk})$ from $\mathcal{E}$ through V_k^* for a challenge for an invalid checkpoint, and the timer Δ has not expired:

1) Remove the wager amount w from V_k^* mainchain account, and use B_{e-1} to verify if the claim is valid.
2) If the claim is valid, invoke a new machine V_l' to replace V_l^* (the machine that is the leader of the epoch), update $\mathcal{V} := \mathcal{V} \backslash \{V_l^*\} \cup \{V_l'\}$, return the wager w to V_k mainchain account, reset the transactions registry $B(\text{id}_{C_i^*}, e) := 0$, $B(\text{id}_{C_j^*}, e) := 0$, $Q_{txn} = \emptyset$ and send $(\texttt{ready})$ to $\mathcal{E}$. Otherwise send $(\texttt{ready})$ to $\mathcal{E}$

B. Upon receiving $(\texttt{turnToTalk})$ from $\mathcal{E}$ through C_i^*, for invalid transaction(s) and the timer Δ has not expired:

1) Remove the wager amount w from C_i mainchain account, and use PP to verify if the claim is valid.
2) If the claim is valid, invoke a new machine V_l' to replace V_l^* and update $\mathcal{V}$, return the wager w to C_i^* , reset the n transactions registries $b_i^s := 0$, $b_j^r := 0$, $Q_{txn} = \emptyset$ and send $(\texttt{ready})$ to $\mathcal{E}$. Otherwise send $(\texttt{ready})$ to $\mathcal{E}$

C. Upon receiving ANY message and the timer Δ has expired:

1) Update $B(\text{id}_{C_i^*}, e) := B(\text{id}_{C_i^*}, e) - x$ and $B(\text{id}_{C_j^*}, e) := B(\text{id}_{C_i^*}, e) + x$, reset $b_i^s := 0$, $b_j^r := 0$, $Q_{txn} = \emptyset$, increase the epoch $e := e + 1$, $j := j + 1$ (the leading registry) and send $(\texttt{ready})$ to $\mathcal{E}$.

Fig. 4. Ideal functionality $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{V}, \mathbb{C})$ for the Sidechain in Vulcan.

The smart contract only accepts messages during the Commit and Pending terms.

(I) **Commit**

A. Upon receiving $(\texttt{commit}, cp_e)$ to $\mathbb{C}$ from any of the validators, V_l:

1) Start a counter for Δ units of time.
2) Send $(\texttt{blockReceived})$ to $\mathcal{E}$, for it to sequentially give the turn to make an operation to clients (deposits, withdraws or challenges) and validators (only challenges), and start the pending term.

(II) **Pending**

A. Upon receiving $(\texttt{depositRequest}, \text{id}_{C_i}, x)$ from a client C_i and if the timer Δ has not expired:

1) Request to $\mathcal{L}$ to transfer x from C_i (in the mainchain), by sending the message $(\texttt{depositRequested}, \text{id}_{C_i}, x)$ to $\mathcal{L}$.
2) If $(\texttt{cannotDeposit}, \text{id}_{C_i}, x)$ is received, send $(\texttt{depositNotReceived}, x)$ to C_i and terminate the operation.
3) If $(\texttt{depositConfirmed}, \text{id}_{C_i}, x)$ is received, send $(\texttt{depositReceived}, x)$ to C_i and to all the validators, for them to update the balance, and terminate the operation.

B. Upon receiving $(\texttt{depositRequest}, \text{id}_{C_i}, x)$ from $\mathcal{E}$ and if the timer Δ has not expired:

1) If $\text{id}_{C_i} \in \mathcal{C}^*$, send $(\texttt{existingClient})$ to $\mathcal{E}$ and terminate the operation.
2) If $\text{id}_{C_i} \notin \mathcal{C}^*$, request to $\mathcal{L}$ to transfer x from C_i (in the mainchain), by sending the message $(\texttt{depositRequested}, \text{id}_{C_i}, x)$ to $\mathcal{L}$.
3) If $(\texttt{cannotDeposit}, \text{id}_{C_i}, x)$ is received, send $(\texttt{depositNotReceived}, x)$ to $\mathcal{E}$ and terminate the operation.
4) If $(\texttt{depositConfirmed}, \text{id}_{C_i}, x)$ is received, send $(\texttt{depositReceived}, x)$ to C_i and to a randomly selected validator.

C. Upon receiving $(\texttt{withdraw}, b, b_{i,e}, h_{B_{e-1}, MP_{txn}})$ from a client C_i and if the timer Δ has not expired:

1) Verify if the provided elements for the proof of possesion are valid. If the verification is valid, send $(\texttt{withdrawNotOK}, b)$ to the client, and to a randomly selected validator. Else, send $(\texttt{withdrawOK}, C_i, b)$ to $\mathcal{L}$ and to a randomly selected validator.
2) Update the total balance $b_t(e) := b_t(e) - b$, and terminate the operation.

D. Upon receiving $(\texttt{exit}, b_{i,e}, h_{B_{e-1}, MP_{txn}})$ from a client C_i and if the timer Δ has not expired:

1) Verify if the provided elements for the proof of possesion are valid. If the verification fails, send $(\texttt{cannotExit,b})$ to the client. Else, update the total balance $b_t(e) := b_t(e) - b$, send $(\texttt{clientExit}, C_i)$ to $\mathcal{L}$ and a randomly selected validator.

E. Upon receiving $(\texttt{noClients})$ from a validator V_k:

1) If the remaining balance $b = 0$, send $(\texttt{executionEnd}, 1)$ to $\mathcal{E}$ and halt.
2) If the remaining balance $b \neq 0$, send $(\texttt{executionEnd}, 0)$ to $\mathcal{E}$ and halt.

F. Upon receiving $(\texttt{challenge}, B_{k,e-1}, w)$ from V_k and if the timer Δ has not expired:

1) Send $(\texttt{getWager}, w, V_k)$ to $\mathcal{L}$, to transfer the wager amount w to $\mathbb{C}$, then use $B_{k,e-1}$ to verify if the claim is valid.
2) If the claim is valid, invoke a new machine V_l' and update the validators set $\mathcal{V} := \mathcal{V} \setminus \{V_l\} \cup \{V_l'\}$, send the message $(\texttt{challengeValid}, V_l, V_l')$ to all the validators and $\mathcal{L}$, keep the epoch equal to e, stop the timer and proceed to the collect term.
3) If the claim fails, update the epoch $e := e + 1$, stop the timer and terminate the operation.

G. Upon receiving $(\texttt{challenge}, PP, w)$ from C_i and if the timer Δ has not expired:

1) Send $(\texttt{getWager}, w, C_i)$ to $\mathcal{L}$, to transfer the wager amount w to $\mathbb{C}$, then use PP to verify if the claim is valid.
2) If the claim is valid, invoke a new machine V_l' and update the validators set $\mathcal{V} := \mathcal{V} \setminus \{V_l\} \cup \{V_l'\}$, send the message $(\texttt{challengeValid}, V_l, V_l')$ to all the validators and $\mathcal{L}$, keep the epoch equal to e, stop the timer and proceed to the collect term.
3) If the claim fails, update the epoch $e := e + 1$, stop the timer and terminate the operation.

H. Upon receiving ANY message and if the timer Δ has expired:

1) Update the epoch $e := e + 1$, and send $(\texttt{epochChanged})$ to a randomly selected validator.

Fig. 5. The smart contract functionality $\mathbb{C}$

.

This section presents the ideal functionality for the protocol. The UCS framework requires that, for the ideal protocol, the functionality of the whole system to be performed by a single entity. To keep protocol transparent to any environment or adversary, the system must still have computing entities representing all the elements that are part of the protocol, but just as "dumb machines" that forward incoming messages to the functionality to perform what was requested, and forward output to the environment $\mathcal{E}$.

A deposit on $\mathbb{SC}$ triggers the deployment of the protocol. We assume that the order of validators passing the leader is following their indexes, e.g., $V_0, V_1, ..., V_n$ and starts over.

Lets denote with $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{V}, \mathbb{C})$ the ideal functionality of the $\mathbb{SC}$, where $\mathcal{V}$ is a set with a prefixed number of validators, $\mathbb{C}$ is a set of contracts deployed on the $\mathbb{MC}$ and $\mathcal{L}(\Delta)$ is the mainchain with delay Δ. The ideal functionality communicates with parties from both $\mathcal{C}$ and $\mathcal{V}$.

To keep the model as simple as possible, we exclude transaction fees in our modeling. The Sidechain functionality is shown in Fig. 4.

We will explain how the interactions happen in four phases:

(I) **Deposit.** Parties can join $\mathbb{SC}$ or increase their balance by depositing an amount x via the message `depositRequest`.

(II) **Withdraw/Exit** Parties from $\mathcal{C}$ can withdraw certain amount of funds or exit the $\mathbb{SC}$ by sending a `withdraw` or `exit` message to the functionality during the pending term of current epoch. After the verification, the functionality will restore to the party's mainchain account the requested amount (or the total balance). In case of an exit message, the requesting party machine will halt.

(III) **Transaction.** The procedure for updating the state of $\mathbb{SC}$ proceeds in epochs which have four distinct terms. We describe three of them in this phase and the other one in other phases. (A) During the *collect term*, parties from $\mathcal{C}$ request transaction trough the `depositRequest` message to the functionality. After the correctness is verified, the states of related parties will be updated accordingly. The functionality keeps n distinct registries, to emulate the fact that there are actually n different validators. Considering the security assumptions, at most $\frac{n}{2} - 1$ of these distinct registries can be corrupted by te adversary. (B) During the propose term, the leader of the current epoch proposes the block via a `proposedBlock` message. In the functionality, this is accomplished by selecting the j-th registry, that corresponds to the leader of the epoch. Then it validates it against the rest of the registries, and if more than half have consistent information, then the block is compiled. (C) The functionality then computes the checkpoint *cp* to start the commit term, upon which it sends the `blockReceived` message to $\mathcal{E}$, for it to start giving turns sequentially to the clients and validators to make deposits, withdraws or challenges.

(IV) **Pending.** Every party can issue a `challenge` message during this term. There two situations where a party should complain. (i) The current block is not committed within a fixed amount of time. (ii) The pending checkpoint is wrong. Specifically, the message of the aggregate signature does not match the hash of the block header, or the aggregate signature is not aggregated by sufficient validators. After receiving at least one challenge message, the functionality will verify with the proof provided by the client or the validator. If the challenge is successful, then the leader will be removed from the committee. Otherwise, the challenger looses its wager.

Now, let us explain how the ideal functionality $\mathcal{F}_{sc}^{\mathcal{L}(\Delta)}(\mathcal{V}, \mathbb{C})$ ensures the security and efficiency properties from Sec. IV-B.

Consensus on sidechain update. Each update in $\mathbb{SC}$ corresponds to a new block produced and confirmed in $\mathbb{MC}$ in the form of a checkpoint. To reach consensus among honest validators, the leader has to propose the block and gather enough votes to commit the block. If the leader does not receive sufficient votes (i.e., less than f votes), it will give up this round and start a new one from the collecting term. After the commitment is confirmed, every party can view the content of the checkpoint by querying $\mathbb{C}$, which means every party can run the verification locally. We assume at least one honest party (including validators and clients) will request a challenge if the checkpoint is faulty. On receiving at least one $\underline{\text{challenge}}$ message, the functionality will run the verification $\overline{\text{algorithm}}$ itself. If the pending checkpoint is detected faulty, the functionality will clear it, remove the leader from the committee, and start a new epoch. Therefore, we can see that this *lazy-challenge* mechanism ensures that every update is agreed among honest parties as long as our assumption (i.e., $n \geq 2f + 1$) holds. Given that each interaction with $\mathbb{C}$ is completed in $O(\Delta)$ units of time and the consensus is achieved in $O(\tau)$ related to the size of $\mathcal{V}$, an update in $\mathbb{SC}$ takes $O(\Delta + \tau)$ units.

Guaranteed withdrawal/exit. A withdrawal or exit can happen only during the *pending term* of each epoch and it will only be handled until the pending checkpoint survives the pending term. If the pending checkpoint is obliterated after the pending term, the end-user has to wait for next round. A withdrawal/exit request can only be handled in the latest checkpoint (the pending one). After the request is approved by $\mathbb{C}$, the requested number of coins will be added to the requester's account within Δ units of time. The correctness of the withdrawal/exit is ensured by the former property since each checkpoint is agreed among honest validators.

D. *The Sidechain Protocol*

The functionalities of the machines that perform the main functions in the protocol, are presented in Fig. 5 and Fig. 6, for the Smart Contract $\mathbb{C}$ and the validators $V_1, \ldots, V_n$, respectively. A brief summary of the protocol is presented now. The reader can refer to Appendix VIII-B, for a detailed description of the UCS model of the protcol.

The protocol starts when the environment machine $\mathcal{E}$ requests a deposit when there is no protocol running yet. Upon this, the smart contract $\mathbb{C}$, the validators $V_1, \ldots, V_n$ and the first client V_1 machines are invoked. After this, more clients can join by requesting deposits to $\mathbb{C}$, which transfers money from the mainchain to the sidechain, for the client to use for internal transactions. The protocol is divided in epochs as described in section III-A, and that division is mirrored in model.

After a client joining the network, it can request to transfer money to other clients, to pay for goods or services in a quick manner. This is made through a request to any $V_l \in \mathcal{V}$. In the collecting term, validators receive all the transactions that the clients request, and store them. Once a certain period has

(I) **Collect**

A. Upon receiving $(\texttt{transferFunds}, (C_i, C_j, b), s_{i,j,b})$ from any client C_i,

1) If $C_i = C_j$, discard and send $(\texttt{ready})$ to $\mathcal{E}$.
2) If $C_i \neq C_j$, verify if $V((C_i, C_j, b), s_{i,j,b}, k_i) = 1$ and $B(\mathrm{id}_{C_i}, e) > b$. If so, set $b_i^s := b_i^s + b$, $b_j^r := b_j^r + b$, $Q_{txn}(e) := Q_{txn}(e) \cup \{tx_l\}$ and $n_{txn} := n_{txn} + 1$. Otherwise, send $(\texttt{txNotValid})$ to the client C_i and send $(\texttt{ready})$ to $\mathcal{E}$.
3) Send $(\texttt{txRequest}, (C_i, C_j, b), s_{i,j,b})$ sequentially to all the other validators.
4) Upon receiving the last $(\texttt{txAck})$ or $(\texttt{txNotAck})$, evaluate if $n_{txn} \geq n_{max}$ or $t_{txn} \geq t_{max}$. If true, send $(\texttt{compile})$ to the leader. Otherwise $(\texttt{ready})$ to $\mathcal{E}$.

B. Upon receiving $(\texttt{txRequest}, (C_i, C_j, b), s_{i,j,b})$ from other validator,

1) If $V((C_i, C_j, b), s_{i,j,b}, k_i) = 1$ and $B(\mathrm{id}_{C_i}, e) > b$, update $b_i^s := b_i^s + b$, $b_j^r := b_j^r + b$, $Q_{txn}(e) := Q_{txn}(e) \cup \{tx_l\}$ and $n_{txn} := n_{txn} + 1$, and send $(\texttt{txAck})$ to the sender. Otherwise send $(\texttt{txNotAck})$ to the sender.

(II) **Propose**

A. Upon receiving $(\texttt{compile})$:

1) Compute $h_{B_{e-1}}$, MT_{txn} and $MT_{\mathcal{P}}$, the tuple B_e (the block), and $s_{B_e,l}$ (the signature).
2) Send $(\texttt{proposedBlock}, B_e, s_{B_e,l})$ to the rest of the validators, sequentially.
3) After the last validator reply (or if the response timed out), proceed to the commit term.

B: Upon receiving $(\texttt{proposedBlock}, B_e, s_{B_e,l})$ from validator V_l

1) Use B_e and $s_{B_e,l}$ to verify the validity of the block. If the check is positive, send $(\texttt{blockApproved}, s_{B_Q}, l')$ to V_l. Otherwise send $(\texttt{blockNotApproved})$.

(III) **Commit**

A. Upon receiving the reply from the last validator for the block proposal (or if the last reply wait timed out):

1) If approval replies are more than $\frac{n}{2}$ compute $H(B_e)$, $QA(H(B_i))$ and *index* and send $(\texttt{commit}, cp_e, \textit{index})$ to the rest of the validators and $\mathbb{C}$.
2) If approval replies are less than $\frac{n}{2}$, set $n_{txn} := 0$, restart the timer $t_{txn} = 0$ and reset the internal parameters for collecting transactions, $b_i^s := 0$, $b_j^r := 0$ and $Q_{txn}(e) = \emptyset$, send $(\texttt{epochRestart})$ to the rest of the validators and $\mathcal{E}$ and send $(\texttt{ready})$ to $\mathcal{E}$.

B. Upon receiving $(\texttt{epochRestart})$,

1) set $n_{txn} := 0$, $b_i^s := 0$, $b_j^r := 0$ and $Q_{txn}(e) = \emptyset$, restart the timer $t_{txn} = 0$ and send $(\texttt{restartAck})$.

(IV) **Pending**

A. Upon receiving $(\texttt{depositReceived}, \mathrm{id}_{C_i}, x)$:

1) If sender is $\mathcal{E}$, if $\mathrm{id}_{C_i} \notin \mathcal{C}^*$, then invoke a machine for that client, update $\mathcal{C}^* := \mathcal{C}^* \cup \{\mathrm{id}_{\mathcal{C}_\rangle}\}$ and $B(\mathrm{id}_{C_i}, e) = 0$, else send $(\texttt{existingClient})$ to $\mathcal{E}$.
2) If sender is any client C_i, update the balances $B(\mathrm{id}_{C_i}, e) := B(\mathrm{id}_{C_i}, e) + x$, and send $(\texttt{endOfTurn})$ to $\mathcal{E}$.

B. Upon receiving $(\texttt{withdrawOK}, C_i, x)$ (or $(\texttt{withdrawNotOK}, C_i, x)$) from $\mathbb{C}$:

1) If $\texttt{withdrawOK}$, update $B(\mathrm{id}_{C_i}, e) := B(\mathrm{id}_{C_i}, e) - x$, and inform the validators, then send $(\texttt{endOfTurn})$ to $\mathcal{E}$, else send only $(\texttt{endOfTurn})$ to $\mathcal{E}$.

C. Upon receiving $(\texttt{clientExit}, C_i, x)$ from $\mathbb{C}$:

1) Update clients set $\mathcal{C} := \mathcal{C} \setminus \{\mathrm{id}_{C_i}\}$.
2) If $\mathcal{C} = \emptyset$, verify if $\mathcal{V} \neq \emptyset$, if so send $(\texttt{clientExit}, C_i, x)$ to a another randomly selected validator and halt, else send $(\texttt{clientExit}, C_i, x)$ to all the validators and then $(\texttt{endOfTurn})$ to $\mathcal{E}$.
3) If $\mathcal{C} = \emptyset$ and $\mathcal{V} = \emptyset$, send $(\texttt{noClients})$ to $\mathbb{C}$ and halt.

D. Upon receiving $(\texttt{turnToTalk})$ from $\mathcal{E}$:

1) If cp_e is invalid, send $(\texttt{challengeCP}, B_{k,e-1}, w)$ to $\mathbb{C}$, otherwise send $(\texttt{endOfTurn})$ to $\mathcal{E}$.

E. Upon receiving $(\texttt{challengeValid}, V_l, V_l')$ from $\mathbb{C}$:

1) Reset the transaction registry, $b_i^s = 0$, $b_j^r = 0$, $Q_{txn}(e) = \emptyset$ to restart the epoch and send $(\texttt{ready})$ to $\mathcal{E}$.
2) If the ID of V_l corresponds to its own ID, send $(\texttt{ready})$ to $\mathcal{E}$ and halt.

E. Upon receiving $(\texttt{epochChanged})$ from $\mathbb{C}$ or a validator:

1) Update the transactions registry $B(\mathrm{id}_{C_i}, e+1) := B(\mathrm{id}_{C_i}, e) + b_i^r - b_i^s$ for all $i \in \{1 \ldots m\}$.
2) If the sender is $\mathbb{C}$, send $(\texttt{epochChanged})$ to the rest of the validators and then send $(\texttt{ready})$ to $\mathcal{E}$.

Fig. 6. Functionality for the Validators $V_1, \ldots, V_n$.

passed, or a certain number or transactions have been accumulated, the propose term begins. In this term one validator takes the lead to compile all the transactions into a block, and propose it. When all the validators have replied to the request for approval (or a fixed timeout has passed), then the commit term begins. In this term, if more than half of the validators approved the block, then the leader generates a QC of the block and submits it to $\mathbb{C}$, after which the pending term begins. If the block fails gaining enough approvals, the protocol returns to the collecting term, and all transactions are discarded. In the pending term, the smart contract is open to receive challenges to the received checkpoint. Also during the pending term, clients can ask for making deposits or withdraws, or to exit. The validators and clients are sequentially queried by $\mathcal{E}$ if they need to request any those operations.

Finally, the protocol execution ends when there are no more clients in the sidechain. When the last client is in the network, and it request to exit, if the withdraw operation for the rest of its balance is successful, the protocol ends and all the machines halt. The details of the security proof is given in Appendix VIII-D.

V. Implementation and Evaluation

In order to evaluate the feasibility and performance of the protocol and the underlying smart contracts, we constructed a simple proof of concept implementation of Vulcan. The smart contract is written in Solidity (around 800 LoC) and deployed in Ethereum Kovan testnet. We also implemented the code of the validator side using Golang (around 1900 LoC) and the client side using Python (around 800 LoC).

An important criteria to evaluate smart contracts running over the Ethereum is the cost of *gas*, which is the fundamental unit of computation in the Ethereum. The fees of any given fragment of programmable computation are universally agreed in terms of gas. Thus we can use the amount of gas to fairly measure the cost of each transaction. Note that we only give run-time cost of our protocol without considering the deployment cost because it is easy to optimize gas costs by extracting all functionality of the smart contract into an external library.

A. Constant Gas Cost

There are two types of transactions, *deposit* and *checkpoint*, having constant gas cost, which means no matter how the number of validators and clients increases, the gas cost of these transactions remains unchanged because they send constant size of data. The exchange rate of gas to ether is determined by GasPrice, which is freely chosen by the sender of the transaction. Higher GasPrice means more incentive for miners to mine that transactions, which leads to shorter latency. In our experiment, we choose an average rate, GasPrice$=4$ gwei (1 gwei $=10^{-9}$ Ether), under which the mean time to confirm a transaction is approximately 20 minutes. We use the exchange rate of 267 between Ether and US dollar[6].

Constant costs of deposit and checkpoint are list in Table I. We observe that it only takes 0.023 USD to issue a deposit and the gas cost 21,912 is slightly more than that of a basic transaction (the gas cost of a basic value-transfer transaction is fixed to 21,000 gas). A checkpoint consists of the hash of the block header (32 bytes), an aggregate signature (65 bytes) and an index number (1 byte). And it takes 272,324 gas (approx. 0.289 USD) for a leader to submit a checkpoint. The duration of an epoch can be adjusted according to the actual use case. For example, a sidechain updates its state hourly (approx. 1 checkpoint for one hour, excluding epoch fails). Then it takes about 7 USD a day, and 2540 USD a year to maintain this sidechain.

TABLE I. Constant costs of deposit and checkpoint.

Type of transactions	Bytes	Gas	ETH (gwei)	USD
deposit	4	21,912	87,600	0.023
checkpoint	98	272,324	1,089,300	0.289

B. Scalability

1) Withdrawal/exit cost: As the size of clients scales up, the cost of a withdrawal/exit increases because the client needs to submit a longer merkle path as the proof of possession (Merkle proof) of the withdrawal/exit. Each withdrawal/exit consists of a key (the address of the requester, 20 bytes), a value (the balance, 4 bytes), a root hash (the hash of the merkle trie, 32 bytes), a branch mask (a “roadmap” to the root node, 1 byte), and an array of siblings (adjacent nodes along the merkle path, 32 bytes). The number of siblings grows with m clients in $O(\log m)$, which is the major cost when the size of the clients becomes large. We measured the withdrawal/exit cost as the size of clients grow in incremental steps (cf. Fig. 7). During each step, we randomly instructed one client to issue a withdrawal/exit transaction to $\mathbb{C}$, recorded the number of siblings included in the transactions and the gas cost.

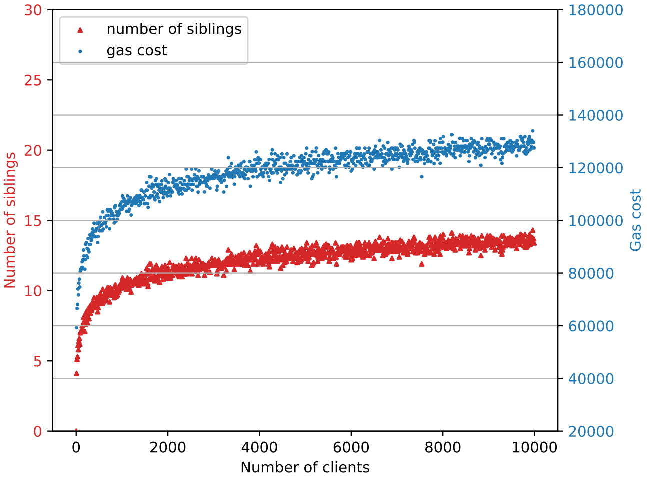

Fig. 7. Gas cost as the number of clients increases.

We can observe that the gas cost grows in the same pattern as the number of siblings grows. During the first 2,000 steps, the gas cost surges to around 80,000 (approx. 17 siblings) and then, for the rest of steps, the speed of growth slows down. When the size of clients came to 10,000, the gas cost only increased to around 90,000 (approx. 20 siblings). Using the

[6] According to the exchange rate as of 31 May, 2019

same GasPrice and the exchange rate, we can calculate that it only costs about 0.1 USD for a user to withdraw/exit in a sidechain network with 10,000 clients. Based on the above observation, we can easily estimate the cost of withdrawal/exit when the size of clients scales up to one million. For example, if one million clients joined the sidechain, a client would include 20 siblings in a withdrawal/exit transaction, which would cost 140,000 gas approximately (approx. 0.15 USD).

2) Aggregate verification cost: Our protocol applies a "lazy-challenge" scheme where $\mathbb{C}$ will not run the verification of the pending checkpoint until some party submits a "challenge" request. There are two types of challenges: (i) a challenge named *challenge-ncp* complaining the leader for not committing the checkpoint, (ii) and a challenge named *challenge-cp* complaining the pending checkpoint (faulty aggregate signature). The former requires parameters including an aggregate signature assembled by the challenger and an index number, while the latter requires no parameters. Both two types of challenges will trigger $\mathbb{C}$ running the verification of aggregate signature. We implemented BGLS aggregate signature [6] in Solidity based on an existing codebase[7].

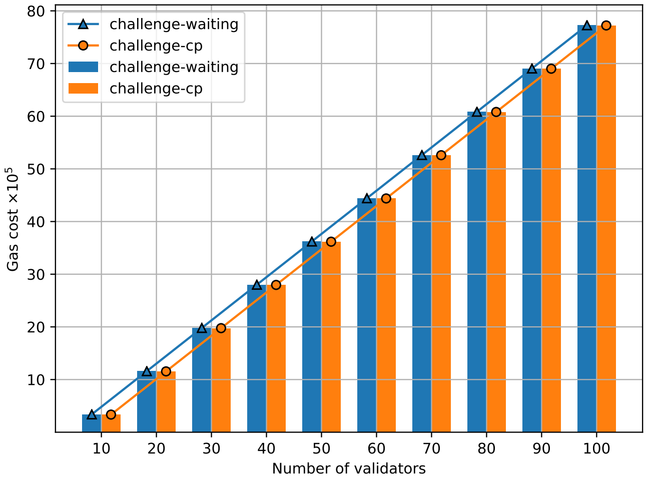

Fig. 8. Gas cost as the number of validators increases.

Fig. 8 showed the gas cost of the aggregate verification as the number of validators increases. We can see that the gas cost (triggered by *challenge-cp*) grows linearly with the validator number from $334,659$ with 10 validators to $7,723,091$, approx. 0.327 USD to 7.546 USD (gas cost triggered by *challenge-ncp* from $338,953$ with 10 validators to $7,727,248$, approx. 0.331 USD to 7.542 USD because of additional parameters). This is because when the number of validators grows, the complexity of the verification increases linearly. Note that the validator number is limited even though we ignore the monetary cost, because there is a *gas limit* which defines all the transactions inside a block allowed to consume in Ethereum. However, the *gas limit* can be adjusted by the miner due to the congestion, which is about $8,000,000$ on average.

[7] https://github.com/Project-Arda/bgls-on-evm

VI. Related Work

The idea of off-chain scaling originated from *Lightening Network* [31], which has emerged as the most promising PCN practice for Bitcoin, followed by *Sprites* and *Raiden* for Ethereum. There exist extensive literature proposing constructions to improve PCN. Revive [21], Spider [33] , and Flash [34] propose dynamic routing algorithms to maximize the throughput and success volume of PCN, while Perun [13] introduces *virtual payment channels* to avoid the involvement of the intermediary for each individual payment, thereby, significantly reducing communication complexity.

In contrast to channels between two parties, sidechain-based approaches allow parties in a sidechain network to freely perform transactions without a prebuilt channel. A first sidechain-based scheme that enables off-chain transactions is Plasma [30], which allows arbitrary consensus algorithms in the sidechain. The consensus mechanism enforces the rules encoded in the smart contract, through which disputes can are resolved fairly. Therefore, as long as the mainchain remains secure, the safety is ensured in the sidechain even if the consensus mechanism is compromised. Two main challenges of current plasma constructions [8][9] are the size of proofs and cumbersome withdrawal/exit procedure.

A very interesting construction similar to ours is called NOCUST [22], which is an account-based *commit-chain*, where a single operator maintains users' account and commits checkpoints at regular intervals. NOCUST is a challenge-response protocol, where users can issue different types of challenges directly to the smart contract to mediate. As such, it requires users to be online at least once within a block time, or a malicious operator might manipulate users' balances. NOCUST also utilize zkSNARK [4] in its smart contract which enables efficient verification of the complete correctness of checkpoints. A major difference between NOCUST and Vulcan is that our protocol features a PoA-based consensus which ensures safety and liveness without online requirement under appropriate security assumption. We argue that by introducing a consensus mechanism over a small-sized committee in the sidechain can mitigate single-point-failure and improve user experience.

Recent literature finds that UCS framework has been widely used to prove the security of off-chain solutions [27][13][17][14][15][25]. Given the serious monetary loss when off-chain protocols are massively applied, we suggest that the security of each off-chain protocol should be formally proved.

VII. Conclusions

In this paper we introduced Vulcan, a sidechain-based off-chain protocol for public blockchains achieve scale-out throughput without sacrificing security. We developed mPoA for validators to achieve consensus in the sidechain which ensures liveness and safety with a maximum of $f < \frac{n}{2}$ validators being Byzantine. We utilized aggregate signature to generate each checkpoint whose validity will only be verified by the smart contract when some party some party. One advantage of Vulcan is that as long as the security assumption holds, clients can always remain offline without worrying assets loss. We defined the security of our protocol in the

UC framework and provided formal proofs. The performance evaluation shows that Vulcan is practical and scalable in terms of monetary cost: around 0.289 USD for each checkpoint and 0.15 USD for each withdrawal in the order of million users. For future work, we plan to extend our off-chain model to support executing smart contracts.

References

[1] L. Aniello, R. Baldoni, E. Gaetani, F. Lombardi, A. Margheri, and V. Sassone, "A prototype evaluation of a tamper-resistant high performance blockchain-based transaction log for a distributed database," in *2017 13th European Dependable Computing Conference (EDCC)*, Geneva, Sep. 2017, pp. 151–154.

[2] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille, "Enabling blockchain innovations with pegged sidechains," 2014. [Online]. Available: http://www.opensciencereview.com/papers/123/enablingblockchain-innovations-with-pegged-sidechains

[3] S. Bag, S. Ruj, and K. Sakurai, "Bitcoin block withholding attack: Analysis and mitigation," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 8, pp. 1967–1978, Aug. 2017.

[4] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, "From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ser. ITCS '12. New York, NY, USA: ACM, Jan. 2012, pp. 326–349.

[5] A. Boldyreva, "Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme," in *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, Jan. 2003, Proceedings*, pp. 31–46.

[6] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "Aggregate and verifiably encrypted signatures from bilinear maps," in *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 2003, Proceedings*, pp. 416–432.

[7] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," in *Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, Dec. 2001, Proceedings*, pp. 514–532.

[8] V. Buterin, "Minimal viable plasma," Jan. 2018. [Online]. Available: https://ethresear.ch/t/minimal-viable-plasma/426

[9] ——, "Plasma cash: Plasma with much less per-user data checking," Mar. 2018. [Online]. Available: https://ethresear.ch/t/plasma-cash-plasma-with-much-less-per-user-data-checking/1298

[10] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, Oct. 2001, Las Vegas, Nevada, USA*, pp. 136–145.

[11] R. Canetti, Y. Dodis, R. Pass, and S. Walfish, "Universally composable security with global setup," in *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, Feb. 2007, Proceedings*, pp. 61–85.

[12] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. E. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer, D. Song, and R. Wattenhofer, "On scaling decentralized blockchains - (A position paper)," in *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, Feb. 2016, Revised Selected Papers*, pp. 106–125.

[13] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, "Perun: Virtual payment hubs over cryptocurrencies," in *2019 2019 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2019.

[14] S. Dziembowski, L. Eckey, and S. Faust, "Fairswap: How to fairly exchange digital goods," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, Oct. 2018*, pp. 967–984.

[15] S. Dziembowski, L. Eckey, S. Faust, J. Hesse, and K. Hostáková, "Multi-party virtual state channels," in *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 2019, Proceedings, Part I*, pp. 625–656.

[16] S. Dziembowski, S. Faust, and K. Hostakova, "Foundations of state channel networks," *IACR Cryptology ePrint Archive*, vol. 2018, p. 320, Aug. 2018. [Online]. Available: https://eprint.iacr.org/2018/320

[17] S. Dziembowski, S. Faust, and K. Hostáková, "General state channel networks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, Oct. 2018*, 2018, pp. 949–966.

[18] J. Eberhardt and J. Heiss, "Off-chaining models and approaches to off-chain computations," in *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers, SERIAL@Middleware 2018, Rennes, France, Dec. 2018*, pp. 7–12.

[19] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," *Commun. ACM*, vol. 61, no. 7, pp. 95–102, Jun. 2018.

[20] E. Gaetani, L. Aniello, R. Baldoni, F. Lombardi, A. Margheri, and V. Sassone, "Blockchain-based database to ensure data integrity in cloud computing environments," in *Italian Conference on Cybersecurity*, Venice, Italy, Jan. 2017.

[21] R. Khalil and A. Gervais, "Revive: Rebalancing off-blockchain payment networks," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, Nov. 2017*, pp. 439–453.

[22] ——, "NOCUST - A non-custodial 2nd-layer financial intermediary," *IACR Cryptology ePrint Archive*, vol. 2018, p. 642, 2018. [Online]. Available: https://eprint.iacr.org/2018/642

[23] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982.

[24] A. Lysyanskaya, S. Micali, L. Reyzin, and H. Shacham, "Sequential aggregate signatures from trapdoor permutations," in *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2004, Proceedings*, pp. 74–90.

[25] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, and M. Maffei, "Anonymous multi-hop locks for blockchain scalability and interoperability," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, Feb., 2019.*

[26] P. McCorry, S. Bakshi, I. Bentov, A. Miller, and S. Meiklejohn, "Pisa: Arbitration outsourcing for state channels," *IACR Cryptology ePrint Archive*, vol. 2018, p. 582, 2018. [Online]. Available: https://eprint.iacr.org/2018/582

[27] A. Miller, I. Bentov, R. Kumaresan, C. Cordi, and P. McCorry, "Sprites and state channels: Payment networks that go faster than lightning," July 2017. [Online]. Available: https://arxiv.org/abs/1702.05812

[28] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[29] J. Niu and C. Feng, "Selfish mining in Ethereum," *CoRR*, vol. abs/1901.04620, 2019. [Online]. Available: http://arxiv.org/abs/1901.04620

[30] J. Poon and V. Buterin, "Plasma: Scalable autonomous smart contracts," *White paper*, 2017.

[31] J. Poon and T. Dryja, "The Bitcoin lightning network: Scalable off-chain instant payments," *See https://lightning. network/lightning-network-paper. pdf*, 2016.

[32] Z. Ren, K. Cong, T. Aerts, B. de Jonge, A. Morais, and Z. Erkin, "A scale-out blockchain for value transfer with spontaneous sharding," in *Crypto Valley Conference on Blockchain Technology, CVCBT 2018, Zug, Switzerland, Jun., 2018*, pp. 1–10.

[33] V. Sivaraman, S. B. Venkatakrishnan, M. Alizadeh, G. C. Fanti, and P. Viswanath, "Routing cryptocurrency with the spider network," in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks, HotNets 2018, Redmond, WA, USA, Nov. 2018*, pp. 29–35.

[34] P. Wang, H. Xu, X. Jin, and T. Wang, "Flash: Efficient dynamic routing for offchain networks," *CoRR*, vol. abs/1902.05260, 2019. [Online]. Available: http://arxiv.org/abs/1902.05260

[35] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

VIII. Appendix

A. Interactive Exit

Interactive exit is to handle situations where (i) the current epoch is in the pending term, but they have not received the block from any of the validators; (ii) the received block does not match the checkpoint; and (iii) the received block can match the checkpoint, but it has conflict with the local view. When participants or honest validators have the above observations, they should initiate an *interactive exit* to compulsively demand the leader to provide $\mathbb{C}$ with sufficient evidence to resolve the dispute. The core idea behind the *interactive exit* is to instruct $\mathbb{C}$ to recalculate the balance of a participant based on that of the last checkpoint (we assume the balance of that participant is correct in the last checkpoint) and the transactions happened during the current epoch and related to that participant.

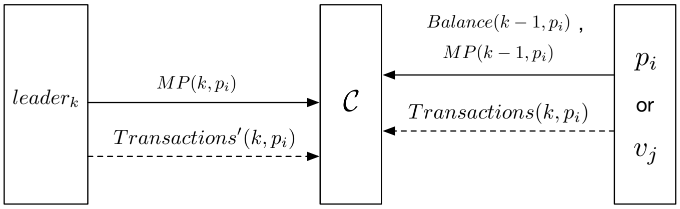

Fig. 9. The procedure of the interactive exit.

A more specific procedure of *interactive exit* is illustrated in Fig. 9. In the pending term of the epoch k, C_i or v_j issues an *interactive exit* by providing $Balance(k-1, C_i)$ and $MP(k-1, C_i)$ (an honest validator and issue the interactive exit in place of C_i since we assume that participants are not always online), which are the balance of C_i enforced at the epoch $k-1$ and the corresponding merkle path. After $\mathbb{C}$ verify the request, it will demand the $leader_k$ to respond to the request with the merkle path of C_i at the epoch k within time Δ otherwise the *mass exit*. If the $leader_k$ successfully submit $MP(k, C_i)$, then within the next time Δ, both of them can send $Transactions(k, C_i)$ to $\mathbb{C}$, where $Transactions(k, C_i)$ refers to a set of transactions issued during the epoch k , related to C_i and signed by both $leader_k$ and C_i. The purpose of this step is that both of them might hide some transactions for their interest. After this step, $\mathbb{C}$ will calculate a new $Balance(k, C_i)$ based on the received evidence. If $Balance(k, C_i)$ cannot match the current checkpoint with $MP(k, C_i)$ provided by the $leader_k$, then the *mass exit* will be triggered. Otherwise, at the end of current pending term, C_i will exit with the $Balance(k, C_i)$ unless someone triggers the *mass exit* before that.

B. UCS Framework Model

The protocol Π is built with 4 computing entities that perform different functions:

- Validators (denoted by $V_1, V_2, \ldots, V_n$)
- Clients (denoted by $C_1, C_2, \ldots, C_m$)
- Global Ledger (denoted by $\mathcal{L}$)

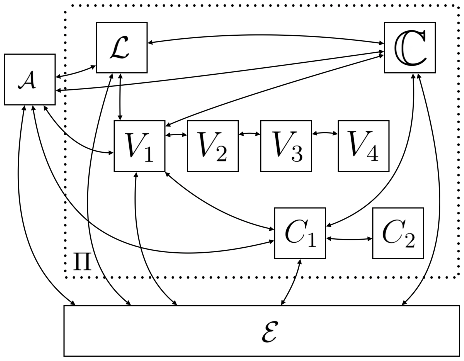

Fig. 10. Protocol Π model.

- Smart Contract (denoted by $\mathbb{C}$)

Each one of these entities are identified with a unique ID. The basic construction of the protocol is depicted in Fig. 10 (only all the communication paths for V_1 and C_1 are depicted for simplicity, showing 2 clients and 4 validators). We denote as $\mathcal{V} = \{V_1, V_2, \ldots, V_n\}$, the set of validators, and $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$ the set of clients.

The following are the functional description of each of the elements listed above. As we describe them, we present all the parameters that they require to work, and the mathematical notation for them.

1) Validators: The validators are nodes that clients request transactions to. They are responsible for collecting and storing the transactions that the clients send, and compile the blocks that containins the accumulated transactions during the epochs, for the other validators to collectively approve.

During the execution of the protocol, validators can take one of two different roles: leader or follower. When a certain amount of transactions are accumulated, or a certain amount of time has passed (parameters that we can assume are fixed prior the start of the protocol), one of the validators take the responsibility of carrying out the process of compiling the transactions into a block. This validator is the leader of the epoch. The rest of the validators are followers, which are only responsible of receiving the block compiled by the leader, and approve it, if the information contained corresponds to its internal registry, or disaprove if not.

The validators parameters are the following. First, there are two read only numbers, that are the maximum amount of transactions per epoch, $n_{\max}$, and the maximum time duration for an epoch $t_{\max}$. There is an index number v_{index}, which has the order number in which the particular validator is picked to be leader, and a number v_n that gives how many validators are in the sidechain. There is also a binary flag v_{status} which indicates the status of the validator, 1 if it is the epoch leader or 0 if is a follower. For syncrhonization purposes, all validators are aware of the current leader by $v_{current}$, this parameter is set to 1 at the beginning of the execution, increases whenever an epoch starts, and goes back to 1 if it goes greater than n. Whenever $v_{current} = v_{index}$, then $v_{status} = 1$, otherwise $v_{status} = 0$. The validators control the number of transactions of the current epoch with n_{txn}, and t_{txn} is a timer with the duration of the current epoch; both numbers are set to 0 at the beginning of the epoch. The validators also hold $\mathcal{P}^* = \{\mathrm{id}_{P_1}, \mathrm{id}_{P_2}, \ldots, \mathrm{id}_{P_m}\}$, the set of IDs of the clients. Also, a function $B : \mathcal{P}^* \times \mathbb{N} \to \mathbb{N}$, that gives the balance of the corresponding client at a any epoch e, i.e., $B(\mathrm{id}_{C_i}, e) = b_{i,e}$, where $b_{i,e}$ is the balance of the client C_i at epoch e. Among the validators, the balances function (which can be viewed as an array), must be equal. We denote individual transactions (plain text transactions) as a tuple, (C_i, C_j, b), where C_i is the sender, C_j is the receiver and b the amount. Actual transactions require a digital signature to ensure that they legitimately come from the client C_i, so the transactions that are sent to validators are also tuples, denoted by $tx_l = (C_i, C_j, b, s_{i,j,b})$, where $s_{i,j,b}$ is the digital signature. They also hold the transactions queue $Q_{txn}(e)$ (for the epoch e), that at the beginning of each epoch is an empty set. As transactions are issued by clients, they are appended to this set, so it is of the form $Q_{txn}(e) = \{tx_1, tx_2, \ldots, tx_l\}$. Lastly, the validators hold b_i^s and b_i^r, that are a pair as accumulators for the received and sent amounts for each client during an epoch. At the beginning of an epoch, both are equal to zero.

2) Clients: The clients are the nodes that join the protocol and make use of $\mathbb{SC}$ to make off-chain money transfers to other clients. When a new client wants to join $\mathbb{SC}$, it has to make an initial deposit to the smart contract, and identify itself with an ID that is not already part of $\mathbb{SC}$. Also, clients that have a positive balance on $\mathbb{SC}$, can withdraw their money to have it returned to $\mathbb{MC}$. When a client that is already part of the network, wants to leave $\mathbb{SC}$, it has to request its exit, which automatically issues withdraw for the total of its remaining balance, then the ITM that represents the client goes to a halt state.

The clients are responsible of keeping track of its own balance, and query from the smart contract the necessary elements (Merkel path) to provide proof of possession, whenever any of them requiere to withdraw from or exit $\mathbb{SC}$.

For this scheme to work, each client requires to have a function $B_i : \mathbb{N} \to \mathbb{N}$ that gives its own balance for any epoch, i.e., $B_i(e) = b_{i,e}$, where $b_{i,e}$ is the balance at epoch e for client C_i. We assume that if a client joined at an epoch k, (i.e., it did not join since the beginning of the execution), then $B_i(k') = 0$ for all $k' < k$.

3) Global Ledger: The Global Ledger $\mathcal{L}$ is the computing entity that represents the functionality of $\mathbb{MC}$. It basically receives the block headers of each of the transaction batches, to serve as checkpoints for transactions occurred on $\mathbb{SC}$. For our purposes, we can assume that clients have infinite off-chain balance, since the security of the mainchain is outside of the scope.

4) Smart Contract: The Smart Contract $\mathbb{C}$ is the computing entity that controls the execution of the protocol by having a counter e that holds the current epoch, this counter starts in 1 at the beginning of the execution. Also, it holds a function that controls the total money that is flowing in the protocol at any epoch, $b_t : \mathbb{N} \to \mathbb{N}$, so $b_t(e) = \sum_i b_{i,e} = \sum_i B(\mathrm{id}_{C_i}, e)$. It also holds the set of IDs for the validators $\mathcal{V}^* = \{\mathrm{id}_{V_1}, \mathrm{id}_{V_2}, \ldots, \mathrm{id}_{V_n}\}$.

The security proof relies on making to any environment, the real protocol indistinguishable from the ideal functionality $\mathcal{F}$ designed to model how the protocol works, and presented in section 4. Figure 11 depicts the ideal protocol.

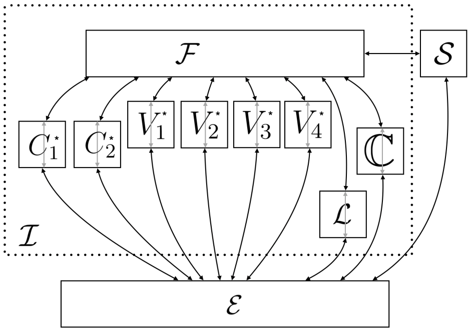

Fig. 11. Ideal protocol $\mathcal{I}$ model.

C. UCS Protocol Evolution

The protocol can be in any of the terms of an epoch, as described on section III-A .

- Collecting Term
- Proposing Term
- Committing Term
- Pending Term

The terms are described below, from the perspective of the protocol model.

1) Collecting Term: The transaction collect is the term in which a clients can request transfer of funds to other clients. This term starts at the execution of the protocol, once the required machines are invoked in Π, or after a block has been successfuly commited and accepted. At the start of the term, the smart contract increases the epoch count, and starts a counter in t_{epoch}. and each validator updates their balance registry as $B(\mathrm{id}_{C_i}, e+1) = b_{i,e} + b_i^r - b_i^s$, sets their accumulators to zero, $b_i^s = 0$ and $b_i^r = 0$, updates the current leader index, $v_{current} + 1$, sets the status flag $v_{status} = 1$ if $v_{index} = v_{current}$, or zero if not, and the set of transactions queue is initialized as empty, $Q_{txn}(e+1) = \emptyset$.

After this, the collecting itself begins. During this, every validator stores a temporary registry of all the transactions sent and received by each client by accumulating the amounts of sent and received money in b_i^s and b_i^r. Also, the transactions queue is assembled. The flow of the transactions requests, is controlled by $\mathcal{E}$. It sequentially sends (`turnToTransfer`) to the clients, so they can request transactions to any of the validators.

For simplicity and for the purposes of the model, we denote the plain text transactions as (C_i, C_j, b), and consider that there are three functions: S (sign), V (verify) and H (hash). The H function returns the hash of the input (if the input is a tuple, it returns hash of the concatenated elements of the tuple), and the other two functions are such that $s_{i,j,b} = S(H(C_i, C_j, b), k_i)$ is the digital signature for the transaction (with k_i being the private key of C_i), and $V((C_i, C_j, b), s_{i,j,b}, \mathrm{id}_i) = 1$ if $s_{i,j,b}$ was indeed created with $H(C_i, C_j, b)$ and k_i, and 0 otherwise, recalling that id_i serves both as the ID and public key of the client C_i.

When C_i wants to transfer money to C_j, and receives the turn to do so, C_i randomly selects a validator V_k, and sends it the message $(\texttt{transferFunds}, (C_i, C_j, b), s_{i,j,b})$. If the sender and receiver are the same, the operation is terminated. This is to avoid the risk of a malicious client to purposely create a loop or a waste network or computing resources. The validator uses $s_{i,j,b}$ and id_{C_i} to verify that the transaction legitimately comes from C_i (if $V((C_i, C_j, b), s_{i,j,b}, \text{id}_i) = 1$), and also verifies that the balance is enough (i.e., $B(\text{id}_{C_i}) - b^s_i + b^r_i \geq b$).

If any of the verifications fail, then V_k sends to the requesting client $(\texttt{txNotValid})$ and the operation is terminated.

If both verifications pass, two things happen: it updates the transaction accumulators as $b^s_i + b$ and $b^r_j + b$, and updates the transactions queue as $Q_{txn}(e) \cup \{tx_l\}$, with the transaction being $tx_l = (C_i, C_j, b, s_{i,j,b})$. Then, V_k sends the message $(\texttt{txRequest}, (C_i, C_j, b), s_{i,j,b}, i)$ to $\mathbb{C}$ for it to increase the transactions count txn, and sequentially to the rest of the validators, so the transaction can be added to their local transactions queue and to increase their local accumulators b^s_i and b^r_i. The sequence to send the transactions to the rest of the validators is such that $v_{current}$ is the last validator that receives the transaction. This is to control the start of the proposing term. The message is also sent. After this, the operation is terminated.

By accumulating the transactions amounts in b^s_i and b^r_i, the validators get the the ease of keeping the actual balances unchanged, in case the block is rejected and discarded.

2) Proposing Term: If the transactions queue $Q_{txn}(e)$ accumulates n_{max} of elements, or if t_{max} units of time pass since the start of the epoch, then the Block Proposal term starts. The validator for which $v_{status} = 1$. It proceeds to assemble the block with all the transactions collected in the round, which are contained in the $Q_{txn}(e)$ set.

For the purposes of the model, this only involves sending the message $(\texttt{proposedBlock}, B_e, s_{B_e,l})$, where B_e is the actual block containing the transactions, and $s_{B_e,l} = S(H(B_e), k_l)$ is the corresponding digital signature of the block, to the rest of the validators.

We consider the block itself as a tuple, whose structure is $B_e = (e, H(h_{B_{e-1}}), cp_{e-1}, R_{txn}, R_a, MT_{txn}, MT_{\mathcal{P}})$, where e is the current epoch, $h_{B_{e-1}}$ is the header of the previous block (and $H(h_{B_{e-1}})$ its hash), cp_{e-1} is the previous checkpoint, R_{txn} is the Merkle root of the transactions tree, R_a is the Merkle root of the accounts tree, MT_{txn} is a n-tuple containing all the entries of the Merkle tree of the transactions, and $MT_{\mathcal{P}}$ is the n-tuple containing all the entries of the Merkle tree of the accounts. Since the first five entries of the tuple are what constitutes the header, the block can be written as $B_e = (h_{B_e}, MT_{txn}, MT_{\mathcal{P}})$.

For the protocol it is necessary that at least half of the group of validators approve the block. All the validators, except for the one executing the transaction, have to reply to the request with either $(\texttt{blockApproved}, s_{B_Q,l'})$, where $s_{B_Q,l'}$ is the signature of the block from follower $V_{l'}$, or $(\texttt{blockNotApproved})$.

If V_k received less than half of approvals, the epoch has to be restarted, first with the leader resetting the epoch transactions by setting $b^s_i = 0$, $b^r_i = 0$ and $Q_{txn}(e) = \emptyset$, and then by sending $(\texttt{epochRestart})$ to the rest of the validators for them to proceed in the same way.

3) Commit Term: The validator V_k collects all the replies, and if more than half of the validators reply with approval, then V_k has to send $(\texttt{commit}, cp_e)$ to $\mathbb{C}$, recalling that $cp_e = (H(B_e), QA(H(B_i)), index)$ is the check point for epoch e, and $QA(H(B_i))$ is the aggregate signature for all the validators that approved the block, created with the signatures $s_{B_Q,l'}$ from each validator $V_{l'}$ that approved the block. V_k also send that message to the rest of the validators and $\mathcal{E}$, for them to be aware that a block has been successfuly proposed to $\mathbb{C}$.

When $\mathbb{C}$ receives the commit message from the validator that is the leader of the epoch, it sends $(\texttt{commitReceived})$ to the participants, and starts a countdown for a fixed amount of time Δ (parameter under control of the smart contract), to wait for challenge messages.

To proceed to the pending term, $\mathbb{C}$ sends $(\texttt{blockReceived})$ to $\mathcal{E}$.

4) Pending Term: The pending period, is the period of time in which several actions can be triggered, by the clients or validators. For this, upon receiving the message $(\texttt{blockReceived})$ by $\mathbb{C}$, $\mathcal{E}$ sequentially request any operation to the validators and to the clients with the message $(\texttt{turnToTalk})$. The order in which validators are requested is the order they take the lead to propose the block, and the validators in the order in which they joined. Upon receiving $(\texttt{turnToTalk})$, the following actions can be triggered.

Deposits. Only can be sent by clients. This is the process by which a client C_i transfers money to be used in $\mathbb{SC}$. For this, a client C_i sends $(\texttt{depositRequest}, \text{id}_{C_i}, x)$ to $\mathbb{C}$. After receiving the request, $\mathbb{C}$ initiates a transaction on $\mathbb{MC}$ to request the amount from C_i. For this, it sends $(\texttt{depositRequested}, \text{id}_{C_i}, x)$ to $\mathcal{L}$. If the client has not enough funds on $\mathbb{MC}$, $\mathcal{L}$ replies to $\mathbb{C}$ with $(\texttt{cannotDeposit}, \text{id}_{C_i}, x)$. Upon receiving the message, $\mathbb{C}$ sends $(\texttt{depositNotReceived}, x)$ to the client C_i and terminates the operation.

If, on the other hand, the client has enough funds on $\mathbb{MC}$, $\mathcal{L}$ transfers the requested amount to $\mathbb{C}$, and replies with the message $(\texttt{depositConfirmed}, \text{id}_{C_i}, b)$. Then $\mathbb{C}$ updates the total balance as $b_t + b$ and sends the message $(\texttt{depositReceived}, \text{id}_{C_i}, b)$ to all the validators for them to update the client's balance as $b' = B(\text{id}_{C_i}, e) + b$ and the balances function B, so that $B(\text{id}_{C_i}, e) = b'$, and to the client C_i, for it to acknowledge the deposit. After this, the operation is finished.

If the machine that sends the message for a deposit is the environment $\mathcal{E}$, it is interpreted as a request from a new client to join $\mathbb{SC}$, provided the requesting ID is not already a client, i.e., $\text{id}_{C_i} \notin \mathcal{P}^*$. In this case, the process that is followed is exactly as the one described previously, with the difference that, when the first validator receives the message to update the balances, and that ID is new to the protocol, a ITM is invoked for that particular ID. If $\mathcal{E}$ sends the deposit request with the ID of an existing client, then the process is terminated. If $\text{id}_{C_i} \in \mathcal{P}^*$, then the validator sends $(\texttt{existingClient})$ to $\mathcal{E}$ and terminates the operation.

For this to work, $\mathbb{C}$ sends the $\texttt{depositReceived}$ message sequentially to the validators in a predetermined way.

Challenges. Since the block content is known for every participant, all of them are aware of what was submitted to $\mathbb{C}$, and therefore anyone can issue a challenge message. If no challenge messages are received within Δ, then this term ends, with the committed block and checkpoint are assumed to be accepted by everyone. When this happens both the epoch counter e and the leader pointer $v_{current}$ are increased in one, and the collect term starts again.

However, if the submitted check point is invalid from the perspective of a validator V_k, (because the submitted block has inconsistencies with the local registry), it sends the message $(\texttt{challenge}, B_{k,e-1}, w)$ to $\mathbb{C}$. B_k is a tuple that contains the information necessary to prove that the current block is inconsistent: the hash of the previous checkpoint, and the aggregate signature of the current block (the one being challenged). The amount w is subtracted from V_k mainchain account and transferred to $\mathbb{C}$, this is by requesting $(\texttt{getWager}, w, V_k)$. After that, $\mathbb{C}$ evaluates the proof $B_{k,e-1}$. If the verification is not successful (the provided proof is not consistent with the submitted block and the hash of the previous check point), then the operation is terminated, and the wager is not returned. If the verification is successful, the validator that lead the epoch is removed from $\mathcal{V}$, $\mathbb{C}$ invokes a new machine to replace it V_l', sends $(\texttt{challengeValid}, V_l, V_l')$ to the validators to reset the transactions registry, $b_i^s = 0$, $b_j^r = 0$, $Q_{txn}(e) = \emptyset$, to restart the epoch.

A very similar process is done, if the challenge comes from a client C_i. The message is the same, $(\texttt{challenge}, PP, w)$, except that for clients, the provided information with the challenge is PP a proof of possesion for the balance of the previous epoch, and the set of transactions from the current epoch. If the challenge is successful, the validator is removed and the wager returned. Otherwise, the wager is lost for the client, although it remains in the sidechain.

Whitdraws. With this, any client can have some amount of its balance, returned to $\mathbb{MC}$. For this, the client C_i sends $(\texttt{withdraw}, b, b_{i,e}, h_{B_{e-1}}, MP_{txn})$ to the smart contract, (b being the amount requested to be withdrew, and MP_{txn} is the Merkle path to be used for the proof of possession). Upon receiving the message, the smart contract is in charge of verifying two things: that the balance is enough and that proof provided is valid.

If the requested amount is not enough (the client's balance is less than the requested amount, $b_{i,e} < b$), or the provided proof of possession is not valid, $\mathbb{C}$ sends to the client C_i the message $(\texttt{withdrawNotOK}, b)$, and the operation is terminated.

If on the other hand, the verifications are passed, $\mathbb{C}$ updates the total balance as $b_t(e) - b$, and sends to the validators the message $(\texttt{withdrawOK}, C_i, b)$, for them to update the balances function as $B(id_i, e+1) = b_{i,e} - b$, and to $\mathbb{MC}$ $\mathcal{L}$, for it to transfer b from $\mathbb{C}$ to C_i on its on chain account. After this, the operation is terminated.

Exit. This is the same as the withdraw process, with the exception that after processing it (if it is successful), then the client is removed from the protocol, by it going to a halt state. For this, the client C_i sends the message $(\texttt{exit}, b_{i,e}, h_{B_{e-1}}, MP_{txn})$ to the smart contract. Then, $\mathbb{C}$ makes the same two verifications as in a regular withdraw. If any of those are not are not successful, $\mathbb{C}$ sends the message $(\texttt{cannotExit}, b)$ to the client, and the operation is terminated.

If the verifications are successful, then $\mathbb{C}$ updates the total balance, $b_t(e+1) = b_t(e) - b_{i,e}$ and sends $(\texttt{clientExit}, \mathrm{id}_{C_i})$ to the validators and the requesting client, for the former to update their clients set as $\mathcal{P}^* \backslash \{\mathrm{id}_{C_i}\}$, and to the latter for the client's $\mathcal{P}_i$ ITM to halt. With this, $\mathcal{P}_i$ cannot be activated again during the execution of the protocol, and its ID cannot be used for a new client. After this, the operation is terminated

What has been described so far, is the general overview of the protocol. However, for a full security proof, we have to create an ideal model that encompass all the functionalities in a single computing entity $\mathcal{F}$.

D. Security Proof

For the security proof we will use as basis the formal definitions of the protocol from section IV, by showing that for any adversary $\mathcal{A}$ interacting with the real protocol (Fig. 10), we can always find a simulator $\mathcal{S}$ such that the environment machine cannot distinguish if it has interacted with the real protocol Π or the ideal protocol $\mathcal{I}$. The concept of distinction of one protocol from the other, comes from a vanishing probability of getting different outputs on the environment machine, after running with Π or $\mathcal{I}$.

A simplification with respect to the full UCS model, is that we do not make use of session IDs, since we consider all interactions to outside entities through the environment $\mathcal{E}$, i.e., no machine from the protocol can communicate to outside machines.

For the proof, we are going to use a constructive approach, this is, we are going to take the computing entities we described in appendix VIII-B, and evaluate how the protocol works interacting with it (as depicted in figure 10)

The UCS model requires that at any point of the execution, only one machine is active, which means the protocol operates with a sequential execution and whenever a machine sends input to or request output from other machine, it stops its execution and the other machine activates upon getting the input or upon being requested to compute some output.

To this end, we will consider that the flow of the execution of the protocol is controlled by the environment as follows. First, the execution starts when $\mathcal{E}$ requests a deposit for a new client, and there are no machines yet (no validators, clients, smart contract or global ledger); the input parameter for the execution is just the amount for this deposit. When this happens, $\mathcal{E}$ invokes the smart contract $\mathbb{C}$ machine. Then $\mathbb{C}$ proceeds to process the deposit, but as there are no global ledger or validators, $\mathcal{L}$ and the set of n validators $\{V_1, \ldots, V_n\}$ are invoked at the moment they are required (n being a fixed predefined parameter by $\mathcal{E}$). When the deposit is confirmed by $\mathcal{L}$, and the first validator is requested to confirm the deposit to the first client V_1, it invokes the machine for it. Whenever a client or a validator is invoked, it generates its own pair of private and public keys.

As was described, new clients are invoked by validators, but are triggered by $\mathcal{E}$, by requesting deposits with IDs that are not in the clients set. The environment, thus, is aware of all the clients that are in the sidechain and the order in which they joined. We consider then, that the environment sequentially, and in the same order, sends messages to the clients asking if they require to make any movement (deposit, withdraw, transaction or exit). With this, we keep the sequential nature of the protocol, while having the scheme working as described before.

We also consider that the protocol execution finishes when there is just one client left and it requests to exit. Lastly, the output of the environment machine is as follows: 1 if the last withdraw to exit is successful (this is, the last client left the sidechain by providing a correct proof of possession for the remaining balance on the smart contract), and 0 if this withdrawal is not successful, since under these circumstances, at some point something went wrong. The output of the environment machine is the value that is written to its output tape when it halts.

Recall the security assumption is that at most $\frac{n}{2} - 1$ validators could be potentially dishonest. First, we consider the case when the protocol running is $\mathcal{I}$ against an adversary $\mathcal{A}$.

The ideal functionality $\mathcal{F}$ keeps n internal registries, one for each validator. Considering the security assumption, only at most $\frac{n}{2} - 1$ of these registries can be tampered by the action of the adversary $\mathcal{A}$, through backdoor messages to $\mathcal{F}$. Now, to avoid testing the protocol against all possible adversaries $\mathcal{A}$, it suffices to make it interact with the "dummy adversary" $\mathcal{D}$, since if a protocol $\mathcal{I}$ UC-emulates another protocol Π with respect to the dummy adversary, then $\mathcal{I}$ UC-emulates Pi against any adversary [10]. The dummy adversary, basically allows $\mathcal{E}$ to takes over the communication links beteween the machines, this is done by acting as a buffer between $\mathcal{E}$ and all clients in the protocol: if $\mathcal{D}$ receives a message from $\mathcal{E}$ to be delivered to any participant, it sends it as a backdoor message to the intended participant. And if $\mathcal{D}$ receives a backdoor message, it delivers it to $\mathcal{E}$. This is, if $\mathcal{E}$ wants to attack any participant, it does it via $\mathcal{D}$, and any response that $\mathcal{D}$ gets, is forwarded to $\mathcal{E}$. This attacks, in general, are messages to modify any of the information flowing in the protocol: transactions, deposits or withdraws amounts.

The internal registries are, however, protected by the security features that the real protocol counterpart: the block is still composed using hashes, and transactions are still signed. A consequence of this, the probability of finishing the execution of the protocol with a zero output is null:

Assuming that at most $\frac{n}{2} - 1$ registries have been compromised by the adversary, the block composing term could be carried out with compromised information. However, the verification process has to check among the n registries, and after finding inconsistencies in more than half, the block would be discarded. This is also true if one or more of the registries have been modified maliciously for the benefit of one or more malicious clients. This feature, prevents double spending, which in turn would cause the protocol to output 0 upon fishing its execution.

Now, we turn our attention to the protocol Π. Lets consider any adversary $\mathcal{A}$. We are going to build a simulator $\mathcal{S}$ in the following way. $\mathcal{S}$ internally will run an instance of $\mathcal{A}$ and another machine, $\mathcal{S}_{\mathcal{D}}$. This machine will serve as interface from $\mathcal{A}$ to send and receive backdoor messages to and from any participant of Π. $\mathcal{A}$ is connected to $\mathcal{E}$ to receive or send messages, and to $\mathcal{S}_{\mathcal{D}}$. Essentially: (1) When $\mathcal{S}$ is sent a message by $\mathcal{E}$, it runs the internal instance of $\mathcal{A}$. If the output message is intended for one of the participants, it is delivered to $\mathcal{S}_{\mathcal{D}}$, and it sends it to that participant as backdoor. If the output message is intended to $\mathcal{E}$, it is delivered to it. (2) When $\mathcal{S}$ is sent a backdoor message by any of the participants, it is delivered to the internal instance of $\mathcal{A}$, and the resulting output is handled in the same way.

With this construction, we have replicated a subset of functions of the dummy adversary: while $\mathcal{D}$ gives full control to $\mathcal{E}$ to perform any kind of attacks, $\mathcal{S}$ encompass only some of the attacks that may be possible, limited by what $\mathcal{A}$ can do.

As in the case of the ideal protocol, the execution is protected by the security properties defined, so the probability of getting a 0 as an execution output is also null. With this, it becomes indistinguishable for $\mathcal{E}$ to detect if the protocol that run, was Π or $\mathcal{I}$.